EXHIBIT B

To the U.S. Patent & Trademark Office
Please stamp the date of receipt of the following document(s) and return this card to us.
Atty Dkt. No: 00P7500US01 Attorney RSK
Applicants: McDaniel, et al
Serial No.: 09/809,155
Filed: March 5, 2001
X IDS/PTO Form 1449 with 14 references
X Certificate of Mailing Dated: June 18, 2002

To the U.S. Patent & Trademark Office
Please stamp the date of receipt of the following document(s) and return this card to us.
Atty Dkt. No: 00P7500US01 Attorney RSK
Applicants: McDaniel, et al
Serial No.: 09/809,155
Filed: March 5, 2001
X IDS/PTO Form 1449 with 14 references
X Certificate of Mailing Dated: June 18, 2002

OIPE
JUN 2 5 2002
JC95
PATENT & TRADEMARK OFFICE

**SIEMENS Corporation**
IPD-West Coast
1220 Charleston Road
P.O. Box 7393
Mountain View, CA 4039

**PATENT APPLICATION**
**ATTORNEY DOCKET NO.: 00P7500US01**

IN THE
UNITED STATES PATENT AND TRADEMARK OFFICE

Applicants: McDaniel, et al.

Application No.: 09/809,155

Filed: March 5, 2001

Title: **PROGRAMMING AUTOMATION BY DEMONSTRATION**

I hereby certify that this paper is being deposited with the United States Postal Service as first class mail in an envelope addressed to: Assistant Commissioner for Patents, Washington, D.C. 20231, on this date.

Date 6-18-2 Signature [illegible]

THE ASSISTANT COMMISSIONER FOR PATENTS
Washington, D.C. 20231

## INFORMATION DISCLOSURE STATEMENT

Sir:

This Information Disclosure Statement is submitted:

[X] under 37 CFR 1.97 (b), or
(Within three months of filing national application; or date of entry of international application; or before mailing date of first office action on the merits; whichever occurs last)

[ ] under 37 CFR 1.97 (c) together with either a:
[ ] Certification under 37 CFR 1.97 (e), or
[ ] A $240.00 fee under 37 CFR 1.17 (p) authorized to be charged to Deposit Account 19-2179. At any time during the pendency of this application, please charge any fees required or credit any overpayment to Deposit Account 19-2179 pursuant to 37 CFR 1.25.
(After the CFR 1.97 (b) time period, but before final action or notice of allowance, whichever occurs first)

[ ] under 37 CFR 1.97 (d) together with a:
[ ] Certification under 37 CFR 1.97 (e), and
[ ] a petition under 37 CFR 1.97 (d) (2) (ii), required with fee under 37 CFR 1.17(i)(1)
(Filed after final action or notice of allowance, whichever occurs first, but on or before payment of the issue fee)

[ ] under 37 CFR 1.97 (i)
[ ] Filed after payment of issue fee, but before grant of patent - No fee or certification required

Applicant(s) submit herewith Form PTO 1449 - Information Disclosure Citation together with copies of patents, publications or other information of which applicant(s) are aware, which applicant(s) believe(s) may be material to the examination of this application and for which there may be a duty to disclose in accordance with 37 CFR 1.56.

[ ] A concise explanation of the relevance of foreign language patents, foreign language publications and other foreign language information listed on PTO Form 1449, as presently understood by the individual(s) designated in 37 CFR 1.56 (c) most knowledgeable about the content that is given on the attached sheet or by the enclosed English-language search report.

[ ] The undersigned hereby certifies under 37 CFR 1.97(e)(1) that each item of information contained in the information disclosure statement was cited in a communication from a foreign patent office in a counterpart foreign application not more than three months prior to the filing of the information disclosure statement; or

[ ] The undersigned certifies under 37 CFR 1.97(e)(2) that no item of information contained in the information disclosure statement was cited in a communication from a foreign patent office in a counterpart foreign application, and, to the knowledge of the undersigned after making reasonable inquiry, no item of information contained in the information

disclosure statement was known to any individual designated in 1.56(c) more than three months prior to the filing of the information disclosure statement.

[X ] Applicant does not believe any fee is due. However, at any time during the pendency of this application, please charge any fees required or credit any overpayment to Deposit Account <u>19-2179</u> pursuant to 37 CFR 1.25. A duplicate copy of this sheet is enclosed.

The filing of this information disclosure statement shall not be construed as a representation that a search has been made or that no other material information exists. Further, the filing of this information disclosure statement shall not be construed as an admission against interest in any manner or as an admission that the information cited is, or is considered to be material to patentability.

It is requested that the information disclosed herein be made of record in this application.

Respectfully Submitted,

Rosa S. Kim

Rosa S. Kim
Attorney for Applicant(s)
Reg. No.: 39,728
Date: 6-18-02
Telephone No.: (650) 694-5330

| Substitute for form 1449A/PTO | | | | *Complete if Known* | |
|---|---|---|---|---|---|
| **INFORMATION DISCLOSURE STATEMENT BY APPLICANT** | | | | **Application Number** | 09/809,155 |
| | | | | **Filing Date** | March 5, 2001 |
| | | | | **First Named Inventor** | McDaniel, et al. |
| | | | | Group Art Unit | Not yet assigned |
| *(use as many sheets as necessary)* | | | | Examiner Name | Not yet assigned |
| Sheet | 1 | of | 2 | Attorney Docket Number | 00P7500US01 |

**U.S. PATENT DOCUMENTS**

| Examiner Initials* | Cite No.[1] | U.S. Patent Document Number | Kind Code[2] (if known) | Name of Patentee or Applicant of Cited Document | Date of Publication of Cited Document MM-DD-YYYY | Pages, Columns, Lines, Where Relevant Passages or Relevant Figures Appear |
|---|---|---|---|---|---|---|
| | | 5,392,207 | | Wilson, et al. | 2/21/95 | |
| | | 5,933,353 | | Abriam, et al. | 8/3/99 | |
| | | 5,576,946 | | Bender, et al. | 11/19/96 | |
| | | 5,696,914 | | Nahaboo, et al. | 12/9/97 | |
| | | | | | | |

**FOREIGN PATENT DOCUMENTS**

| Examiner Initials* | Cite No.[1] | Foreign Patent Document Office[3] | Number[4] | Kind Code[5] (if known) | Name of Patentee or Applicant of Cited Document | Date of Publication of Cited Document MM-DD-YYYY | Pages, Columns, Lines, Where Relevant Passages or Relevant Figures Appear | T[6] |
|---|---|---|---|---|---|---|---|---|
| | | | DE 197 15 494 A1 | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |

| Examiner Signature | | Date Considered | |
|---|---|---|---|

*EXAMINER: Initial if reference considered, whether or not citation is in conformance with MPEP 609. Draw line through citation if not in conformance and not considered. Include copy of this form with next communication to applicant.

[1]Unique citation designation number. [2]See attached Kinds of U.S. Patent Documents. [3]Enter Office that issued the document, by the two-letter code (WIPO Standard ST.3). [4]For Japanese patent documents, the indication of the year of the reign of the Emperor must precede the serial number of the patent document. [5]Kind of document by the appropriate symbols as indicated on the document under WIPO Standard ST. 16 if possible. [6]Applicant is to place a check mark here if English language Translation is attached.

| | | | |
|---|---|---|---|
| Substitute for form 1449A/PTO | | | |
| **INFORMATION DISCLOSURE STATEMENT BY APPLICANT** | | ***Complete if Known*** | |
| | | **Application Number** | 09/809,155 |
| | | **Filing Date** | March 5, 2001 |
| | | **First Named Inventor** | McDaniel, et al. |
| | | Group Art Unit | Not yet assigned |
| *(use as many sheets as necessary)* | | Examiner Name | Not yet assigned |
| Sheet 2 | of 2 | Attorney Docket Number | |

**OTHER PRIOR ART - NON PATENT LITERATURE DOCUMENTS**

| Examiner Initials* | Cite No.[1] | Include name of the author (in CAPITAL LETTERS), title of the article (when appropriate), title of the item (book, magazine, journal, serial, symposium, catalog, etc.), date, page(s), volume-issue number(s), publisher, city and/or country where published. | T[2] |
|---|---|---|---|
| | | ActiveX Controls – Microsoft Papers, Presentations, Web Sites, and Books, for ActiveX Controls" see http://www.microsoft.com/tech/active.asp | |
| | | Farm, LUEY Visual Language, "Software Farm Creating graphics, user interface and diagramming tools, frameworks, and applications in Java™, see http:www.swfm.com | |
| | | Microsoft Corp., Visual Basic, "Visual Studio Datasheet", see http://msdn.microsoft.com/vstudio/prodinfo/datasheet/default.asp | |
| | | Object Management Group, Common Object Request Broker Architecture (COBRA), "Object Management Group Home Page," http://www.corba.org | |
| | | Siemens SIMATIC WinCC Human Machine Interface, "SIMATIC Interactive Catalog" http://www.ad.siemens.de/simatic/html 76/produkte/index/html | |
| | | WITTEN, "A Predictive Calculator" Watch What I Do: Programming by Demonstration, Edited by Allan Cypher, MIT Press, Cambridge, Massachusetts, 1993. | |
| | | WOLBER, "Pavlov: Programming by Stimulus-Response Demonstration" (Human Factors in Computing Systems, Proceedings SIGCHI 1996, Denver, Colorado, April 1996, pp. 252-259. | |
| | | Smith, David Canfield, "Pygmalion: A Creative Programming Environment", Chapter 1 of Watch What I Do: Programming by Demonstration, ed. Allen Cypher, The MIT Press, 1993, pp. 19-47. | |

| | | | |
|---|---|---|---|
| Examiner Signature | | Date Considered | |

*EXAMINER: Initial if reference considered, whether or not citation is in conformance with MPEP 609. Draw line through citation if not in conformance and not considered. Include copy of this form with next communication to applicant.

[1]Unique citation designation number. [2]Applicant is to place a check mark here if English language Translation is attached.

# ActiveX Controls

ActiveX® controls are among the many types of components that use COM technologies to provide interoperability with other types of COM components and services. ActiveX controls are the third version of OLE controls (OCX), providing a number of enhancements specifically designed to facilitate distribution of components over high-latency networks and to provide integration of controls into Web browsers. These enhancements include features such as incremental rendering and code signing, to allow users to identify the authors of controls before allowing them to execute. Follow the links below to learn more about ActiveX.

Articles in the Press
A listing of media coverage on ActiveX and related technologies from various publications.

White Papers
A listing of technical white papers, FAQs and other documentation on ActiveX and related technologies.

Presentations
A collection of Microsoft® PowerPoint® slides that provide an overview of ActiveX.

Web Sites
A listing of other Microsoft and external developer Web sites that offer technical information, resources and training on ActiveX and related technologies.

Books
A collection of noteworthy books on ActiveX and related technologies.

*Last Updated: 3/30/99*

*Java for your Eyes*

Consulting - Mica Graphics Framework - MiUML Diagram Editor
Philosophy - Company Information

**Announcements!**

- MiUML Release 0.98 (7/1/2001): Modernizing release.
- Mica Release 1.2.1 (7/1/2001) Major bug fix release

**Java Magazines**

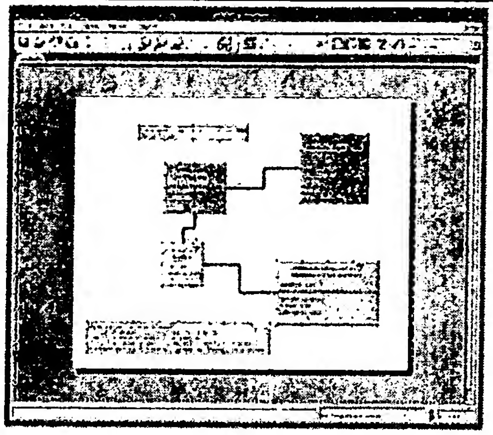
- Java World
- Java Developer's Journal
- Digital Cat's
- Java Toys
- Java Boutique
- Developer Life
- Web Review
- Web Developer

**Java Code Repositories**

- Jars
- Gamelan
- Java Repository
- Java Channel
- Java Programming Resources
- Java Shareware
- Java Software.com
- Java Open Source

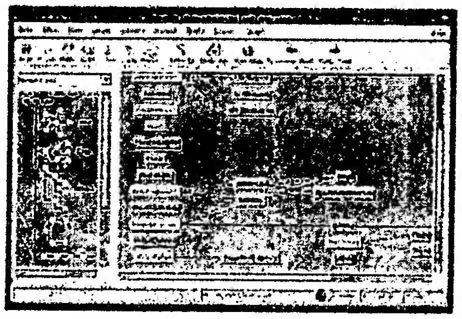
**Free Java Graphics Toolkits**

- Mica Graphics Framework
- Shikotan Graphics
- GEF
- VGJ



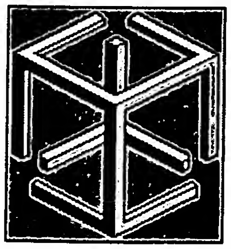
**A UML (Unified Modeling La and Professorial Assistant writ code.**

**Mica**
**Java Graphics Toolkit**



**A Java(TM) Object-Oriented 2 Interface Framework. Include graphics editors. Free with sou**

**Software Federation**
**Consulting/Outsourcing**



**Software Farm's consulting pa able to assist you in all phases requirements and design cons development outsourcing.**

**Cadabra**
**Java MetaFramework**

- EZD
- GFC
- Diva
- SGraphics

## Free Java Widget Toolkits

- Mica Graphics Framework
- Swing
- IFC
- SubArctic
- BISS AWT
- Bongo
- GWT

## Java Graphics Toolkits ($)

- ILOG JViews
- GLG Toolkit
- GO++

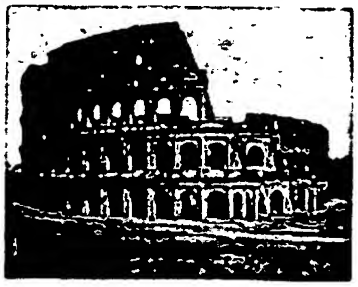
## Java Toolkit Links

- GUI Toolkits/Frameworks
- Graph-Drawing Tools

## Conference Reports

- Our 1999 JavaOne Trip Report

## Cadabra
**Java MetaFramework**

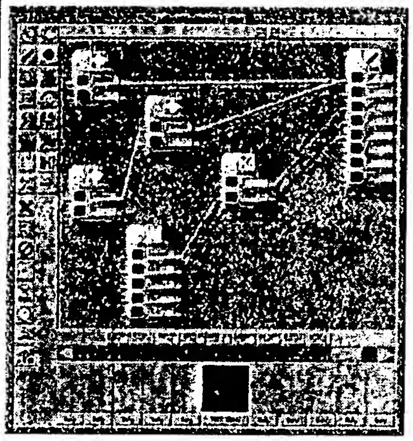


**A Java(tm) Runtime-configura Application MetaFramework. Overview.**

## ZAP
**Java Debugger**

**A Java(tm) Command-line De similar to the Unix(tm) dbx. T debugger API which is too bug This needs to be ported to the Documentation(32K). Download (source and .classes)**

## LUEY
**Visual Language**



**A visual language (interactive assignment of time-dependent graphical data. Written in the language and C++.**

## EditorObject
**C++ Graphics Toolkit**

**An object-oriented graphics li easy creation of 2D vector gra Documentation(1381K). Source(431K). VFACE Visual Framework W**

## VisualADE
**Development Framework**

The *Visual Application Develop* combination of the Editor Obj Cadabra. VisualADE generate textual description files. Writt Examples. Documentation(806K). White Paper(216K).

**PGL**
**Portable Graphics Toolkit**

The *Portable Graphics Library*, hierarchical display lists, a win graphics on the X Window Sys SunView and many PC DOS v Documentation(1349K). Source(1122K)

**Print Utility**
**X11 Screen Capture**

A utility that can be used to sn screen, view, modify and arran preview window, and then pri Encapsulated PostScript or H Documentation.

Microsoft Visual Studio .net

Visual Studio Home | Visual Studio Worldwide

Search This Site

Advanced Search

- Visual Studio Home
- Product Information
- How to Buy
- Technical Resources
- Downloads
- Support
- Community
- Partners
- Developer Tools

Visual Studio .NET Product Information

# Product Overview for Visual Studio .NET

Content Updated: June 04, 2002

Introducing Visual Studio .NET; visionary yet practical, the single comprehensive development tool for creating the next generation of applications has arrived. Developers can use Visual Studio .NET to:

**RELATED LINKS**

- Visual Studio .NET Academic Fast Facts
- Order Visual Studio .NET posters

- Build the next-generation Internet.
- Create powerful applications fast and effectively.
- Span any platform or device.

Visual Studio .NET is the only development environment built from the ground up for XML Web services. By allowing applications to share data over the Internet, XML Web services enable developers to assemble applications from new and existing code, regardless of platform, programming language, or object model.

Visual Studio .NET is available in the following editions:

| Enterprise Architect | Enterprise Developer | Professional |
|---|---|---|
| Visual Studio .NET Enterprise Architect enables you to take advantage of the industry's leading development tool and create sound architectural guidance for development teams. | Visual Studio .NET Enterprise Developer provides a powerful, enterprise team development platform for rapidly building XML Web services and applications. | Visual Studio .NET Professional enables you to rapidly build next-generation XML Web services and applications that target any Internet device and integrate across programming languages and operating systems. |

Ten years ago Visual Basic 1.0 revolutionized application development for millions of developers. Today, Visual Studio .NET sets the stage for the next decade.

Top

Contact Us E-Mail This Page

Vendor Commitments
Standards Based On Specs.
C O R B A Success Stories
C O R B A® For Beginners
General OMG Information
Submit Your Success Story
Submit Your Vendor Commitment
CORBA

OMG Home Vendor Commitments Standards Success Stories Getting Started

Submit success story Submit vendor commitment

**Industrial Automation Systems SIMATIC**

SIEMENS

Home Sitemap Print

Products & Solutions | News Center | e-commerce | Support | Search

Automat
→ Produ
→ Locati
→ Jobs a
Li

Added
Totall Auto
Comp Auto
Safety

Refer
MOB syste
Motio SIMO
SITO
SIWA syste
Syste TOP c

→ SIMATIC® Controllers
→ SIMATIC® Programming Devices
→ PC-based Automation
→ SIMATIC® Industrial PC
→ SIMATIC® Software
→ SIMATIC® Process Control System PCS7
→ SIMATIC® Distributed I/O
→ SIMATIC® Machine Vision
→ SIMATIC® Human Machine Interface
→ Industrial Communication with SIMATIC NET®
→ Siemens Automation Solution Provider
→ SIMATIC® Micro Automation
→ SIMATIC® based Technology

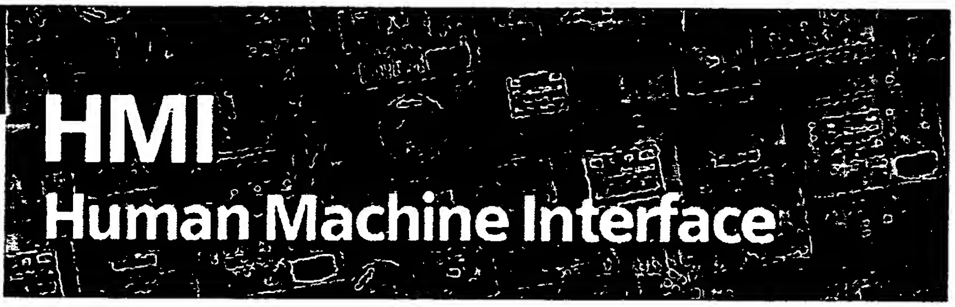
Webmaster Contact

Home | Sitemap | Print

. Products & Solutions . News Center . e-commerce . Support

Search

**Operator Control and Monitoring Systems SIMATIC HMI**



Auto
. Produ
. Industr
. Countr

**Welcome to SIMATIC HMI**

Panels up-to-date

The complete range - All Panels from Micro Panel up to the powerful Multi Panels in one overview for download

SIMATIC Mobile Panel 170 - Flexibility in HMI of machines and plants Product announcement

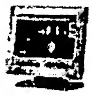
Multi Panel MP 270B – Innovating the class of Multifunctional platforms . Product announcement

New Software options for the Multi Panel MP 370: SIMATIC WinAC MP, the Soft-PLC under Windows CE. Product announcement

---

Industrial LCD Displays up-to-date

Simply brilliant in function and design - 18" LCD as desktop device also with Touch functionality.



---

Software up-to-date

All new features of WinCC Version 5.1 and the new Options for IT-integration in one overview for download

Configuration software SIMATIC ProTool Version 6.0
Taking full advantage of Totally Integrated Automation

Visualization software SIMATIC ProTool/Pro Version 6.0.
conveniant configuration and enhanced runtime functionality

---

Success stories up-to-date

Proper temperature for a Bicycle helmet. SIMATIC TP 170A visualizes heating areas of vacuum molding machine ...

For a beer with a traditional touch - Benedictines at Klosterbrauerei Andechs visualize yeast propagation plant with MP370..

FDA validated hydrogenation plant with exemplary flexibility at Sigma Aldrich in Gillingham UK due to WinCC...

---

Contact

# Pavlov: Programming By *Stimulus-Response* Demonstration

**David Wolber**
Department of Computer Science, University of San Francisco
2130 Fulton St., San Francisco, CA., 94117-1080
(415) 666-6451
wolber@usfca.edu

**ABSTRACT**
Pavlov is a Programming By Demonstration (PBD) system that allows animated interfaces to be created without programming. Using a drawing editor and a clock, designers specify the behavior of a target interface by demonstrating stimuli (end-user actions or time) and the (time-stamped) graphical transformations that should be executed in response. This stimulus-response model allows interaction and animation to be defined in a uniform manner, and it allows for the demonstration of *interactive animation*, i.e., game-like behaviors in which the end-user (player) controls the speed and direction of object movement.

**KEYWORDS**
End User Programming, UIMS, Programming By Demonstration, Programming By Example, Animation

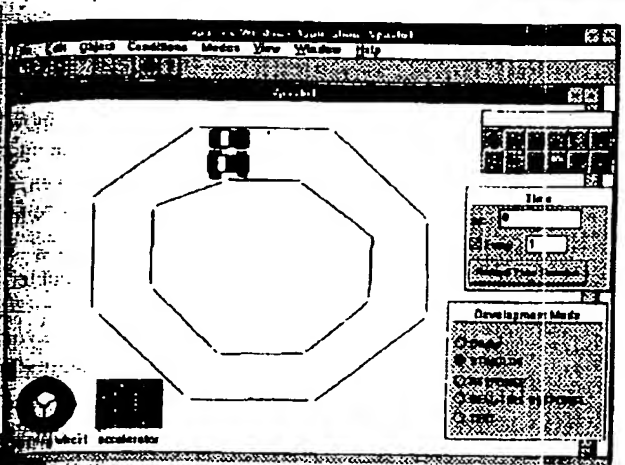
## INTRODUCTION
A visitor to our planet might deduce that most computer users have the necessary skills to quickly and easily graphical user interfaces (GUIs). First, computer users know what they want: any user of today's popular applications is now quite capable of delivering a detailed (and passionate!) discussion on the strengths and flaws of computer interfaces. Second, most computer users have the mechanical skills required to demonstrate the appearance and behavior of an interface: anyone that has used a drawing editor knows how to draw objects with a computer, click on them, and transform them.

But today's development tools have yet to fully tap the potential of the computer user. Though interface builders like Visual Basic have significantly decreased the time and expertise necessary to build *standard* interfaces, the development of more graphical, animated interfaces is still mostly performed by skilled programmers. This time-consuming and costly development is particularly unfortunate given the exploding demand for computer games, interactive entertainment, and animation in even "standard" interfaces, and the recognized importance of more end-user participation in designing interfaces. There has been some progress: an entire book has been published describing research systems that allow interfaces to be created or extended by demonstration rather than programming [2] ; commercially, tools like Macromedia's *Director* allow non-programmers to develop animation and some interaction.

But none of these systems cohesively combine interactive techniques for specifying end-user interaction, graphical transformation, and timing, the three primary ingredients of an animated interface. *Director* is powerful for specifying transformation and timing, but designing simple interaction requires some programming, and more complex interaction requires an expert. The Programming By Demonstration (PBD) systems in [2] present powerful techniques for specifying transformation and some interaction, but do not provide the timing mechanisms necessary for animation.

This paper presents Pavlov, a GUI development system based on the *stimulus-response* model. Stimulus-response provides a cohesive model for demonstrating interaction, transformation, and timing. The model seeks to minimize the cognitive dissonance between concept and design by allowing designers to demonstrate the behavior of an interface exactly as they think of it: "When I do A, B occurs", or "two seconds after the start of the program, this animation begins." Beginning with a blank *target interface*, tabula rasa if you will, the designer uses a drawing tool to draw the interface, then uses the same tool and a clock to demonstrate stimulus-response pairs. In essence, the designer teaches the system in a way that is intuitive to humans:

> The basic physiological function of the cerebral hemisphere throughout the subsequent individual life consists in a constant addition of numberless signaling conditioned stimuli to the limited number of in-born unconditional stimuli, in other words, in constantly supplementing the unconditioned reflexes by conditioned ones.
>
> Ivan Petrovich Pavlov



gure 1. Pavlov Development of a Driving Simulator

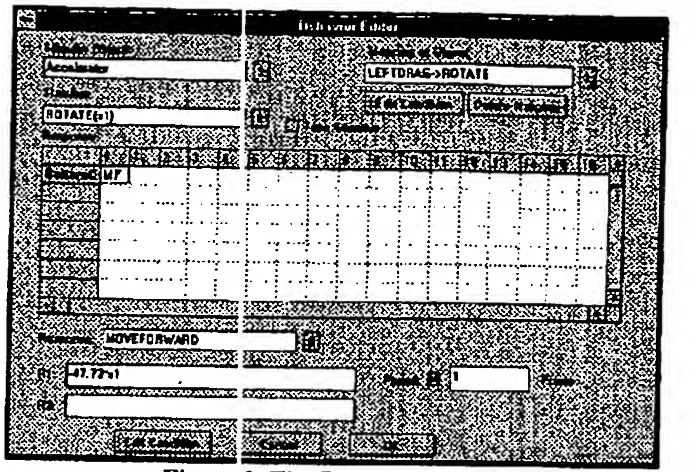


Figure 2. The Pavlov Editor

is stimulus-response model was first used in the thor's DEMO system [13] to demonstrate *non-animated* face behaviors. The model has been extended in vlov so that an interface can be taught about time, odic activity, and the inherent *direction* of some ects. These extensions allow animation as well as action to be designed *within the stimulus-response ework*. It is this intersection between animation and action, not animation per se, that distinguishes Pavlov m other PBD systems. Because of it, designers can onstrate most behaviors that combine interaction and ation, including game-like behaviors in which the user (player) controls the speed and direction of object vement.

## RIVING SIMULATOR EXAMPLE

*e driving simulator, the top car begins moving when rogram begins, follows a pre-defined path around the then stops near its starting point. The bottom car moving only when the driver rotates the lerator". Its speed and direction is controllea by the (the end-user) manipulating the accelerator and the ng wheel.*

e 1 shows the Pavlov environment during opment of the driving simulator (also see the Video in the CHI 96 Video Program). The basic tools are awing editor in the top-right corner, the clock le-right), and the Development Mode Palette (lower- The designer uses the development modes to inform tem as to whether s/he is just drawing the interface mode), demonstrating an end-user or time stimulus ulus mode), demonstrating how the system should d to a stimulus (Response or Real-Time Response or testing an interface (Test mode). The designer clock to demonstrate when an operation should be d (using the top *At:* box), or if an operation should uted periodically (the middle *Every:* box). The utton on the clock, labeled *Record Time Stimulus*, allows the designer to specify that the following responses should be triggered by time.

Another important part of the environment is the editor, shown in Figure 2 This editor displays a textual description of the interface being designed. It serves to provide feedback to a designer as demonstrations are performed, and it allows the designer to modify the behaviors "learned" by the system when necessary. Details of this editor are provided later in the paper.

The first task in creating the Drawing Simulator is to draw the two cars, the road, the accelerator, and the steering wheel. To do so, the designer selects Draw mode from the Development Mode Palette, and makes use of the graphic primitives and grouping mechanisms available in the drawing palette. He also provides names to the drawn objects for later reference.

Next, the designer begins specifying the behavior of the interface. Because s/he wants the two objects shaped like cars to move as cars do, s/he selects each, chooses Object I Set Direction from the menu and enters an angle that defines the *direction attribute* of the particular car. In the simulator, both cars initially face straight ahead on the x-axis, so the designer sets the angle to 0. A vector emanating from its center appears on each car to signify that the car will only move forward and backward in relation to its direction, and must be rotated to change direction. The vector does not appear during execution of the target interface.

The designer is now ready to demonstrate the stimulus-response behavior of the top car. In this case, the stimulus is time: at time 0 (the beginning of execution) the car should begin moving. To demonstrate, the designer selects Stimulus mode, sets the clock *At:* box to 0, and clicks on the Record Time Stimulus button. The system reports that a *time stimulus* has been recorded, and automatically switches to Response mode. For this behavior, the designer wants to demonstrate a special kind of response, a real-time response, so that mode is selected. The designer then selects the move icon in the drawing palette and drags the car around the track. As s/he drags the car, it doesn't ever move diagonally, but instead moves forward towards its nose, and rotates its base (turns) in order to follow the mouse around the track. After the designer releases the mouse button, s/he sees from the editor that the system has recorded a series of discrete, time-stamped responses, made up of alternating MoveForward and Rotate commands.

Next, the designer enters Test mode to visually test the demonstrated behavior. Immediately, the top car begins moving and follows the path that was demonstrated. The designer knows s/he could edit the recorded responses in the editor to modify the path, but for now s/he is satisfied with the behavior of the top car.

The designer then turns his attention to the bottom car. The bottom car's behavior is not triggered by time, but by an end-user action. Thus, instead of demonstrating a time stimulus, the designer plays the role of the end-user and demonstrates an action. After entering Stimulus mode, s/he selects the Rotate icon in the drawing palette, presses the left-mouse button on the rectangle denoting the accelerator, and rotates it clockwise some amount, say -0.36 radians. The system reports that a stimulus was recorded and automatically switches the development mode to Response. The system also records an implicit stimulus-response descriptor mapping the physical action used to the higher-level operation:

(1) On Accelerator.LeftDrag --> Accelerator.Rotate

At this point, the designer needs to demonstrate that the rotation of the accelerator should cause a response of accelerating the movement of the bottom car. First, s/he enters 1 in the *Every:* box of the clock. S/he knows that this will cause the upcoming demonstrated response to be executed periodically every time frame in the target interface. Next, s/he moves the car some amount, say 17 units. Because the car is a "directed" object, the car's movement is restricted: it can only be moved on the vector defined by its direction arrow (note that in Real-Time Response mode this vector is allowed to change). This restriction is as the designer desired: in response to the rotation of the accelerator, s/he wants the car to *move forward*, not change direction. The system reports the recorded stimulus-response descriptor containing a proportional constant (-47.22 = 17/-.36)

(2) On Accelerator.Rotate(s1)-->
BottomCar.MoveForward(-47.22*s1) At 0 Every 1

Next, the designer enters Test mode to check his work. The top car immediately begins its path. The designer, playing the role of the end-user, rotates the accelerator. The bottom car begins moving, and continues to move even after the designer releases the mouse from the accelerator. As the car leaves the right side of the screen, it reappears on the left. The designer again experiments with rotating the accelerator and notices that rotating it clockwise speeds up the car, while rotating it counter-clockwise slows it down.

The designer is nearly satisfied but thinks the accelerator is a little sensitive. He enters the editor (see Figure 2) and selects "Accelerator" as the stimulus object. The stimulus "Rotate(s1)" appears in the box labeled stimulus, and a single response appears in the first cell of the score row labeled BottomCar. The response box below the score contains the response "MoveForward". Its parameter, as in descriptor (2) above, is -47.22*s1. The period box contains "1". To reduce how much the car speeds up in response to the rotate, the designer changes the proportional factor from -47 to -30.0. (alternatively, the period could have been increased). When s/he re-enters Test mode, s/he is satisfied to see that the accelerator is indeed less sensitive.

The next task is to specify the behavior of the steering wheel so the end-user can control the direction of the bottom car. The designer enters Stimulus mode and rotates the steering wheel. Then, in Response mode, s/he sets the *Every:* box in the clock to 1, and rotates the bottom car. The following descriptor is recorded:

(3) On Wheel.Rotate(s1)-->
BottomCar.Rotate(0.25*s1) At 0 Every 1

To test this new behavior, the designer once again enters Test mode. The top car immediately begins its path. The designer rotates the accelerator to get the bottom car moving, then releases the accelerator and rotates the steering wheel to control its direction. He is pleased to note that s/he was correct to set the *Every:* box before demonstrating the rotation of the bottom car: just like a real one, the car continues to turn if the steering wheel remains rotated from its original setting.

The designer continues to test the interface, and soon realizes that if s/he rotates the accelerator counter-clockwise past its origin, the car begins moving backward. To alleviate this problem, s/he uses the editor to delete the previously recorded accelerator behavior, and then re-demonstrates it. First, in Draw mode, s/he sets the top-left point of the accelerator on the left edge of the enclosing rectangle and chooses *Conditions* | *Generate* from the menu. Then s/he demonstrates the stimulus of rotating the accelerator. A dialog appears listing a set of graphical conditions relating the stimulus object to other objects in the interface. The designer selects the condition "Accelerator.Within(EnclosingRectangle)", and the system records a modified version of the originally recorded stimulus-response descriptor (1).

(4) On Accelerator.LeftDrag --> Accelerator.Rotate
When Accelerator.Within(EnclosingRectangle)

The designer proceeds to demonstrate the response of moving the car forward, as s/he did in the first iteration. Afterwards, s/he re-enters Test mode, and is satisfied to see that s/he (playing the role of the end-user) is restricted from rotating the accelerator outside its enclosing rectangle. Since the top-left point of the accelerator begins on the left edge of the enclosing rectangle, there is no way to rotate it counter-clockwise past its origin, so the car cannot move backward.

## THE STIMULUS-RESPONSE MODEL

The driving simulator example illustrates many features of the stimulus-response model, including the extensions that allow interactive animation to be defined. This section describes the model in more general terms in order to 1) explain the inferences made by the system in the example, and 2) bridge the gap between the specific and the general, i.e., persuade the reader that Pavlov is useful for designing all kinds of interfaces, not just driving simulators.

An interface is viewed as a stimulus-response machine. Stimuli are either physical actions (e.g., drag the mouse while pressing the left-button), higher level operations, or time. The interface responds to stimuli by executing a set of time-stamped operations. Operations either create, transform, or delete objects. The set of operations includes the primitives found in most drawing editors and one additional primitive, *move forward*. This additional primitive allows an object to be moved while constrained to the vector defined by its direction attribute. Together, the operations offer the basic functionality necessary to demonstrate nearly all animated interface behaviors.

### The Semantics of Stimulus-Response

The challenge of a stimulus-response development system is to provide clear syntax and semantics for how the designer uses the set of physical actions and operations to demonstrate the behavior of the target interface. The "syntax" of Pavlov is straight forward: the designer changes development modes to inform the system whether his intent is to draw, demonstrate a stimulus or response, or test the interface.

Providing clear semantics is a more challenging problem: the goal is for the system to always record a stimulus-response descriptor that perfectly matches the intent of the designer's demonstration (this is the primary challenge of all PBD systems). Pavlov uses an *explanation-based learning* approach: from a single demonstration of a stimulus-response pair, the system uses domain knowledge and the information provided by the demonstration to record as reasonable a stimulus-response descriptor as possible. If necessary, the designer can then use Pavlov's powerful editing facilities to modify the descriptor.

This simplistic strategy differs from other systems that allow a designer to refine behavior descriptions through multiple demonstrations. Such an *empirical-based learning* approach allows more complex behavior to be specified, but complicates the semantics of the system.

### The Semantics of a Stimulus Demonstration

In Stimulus mode, the designer demonstrates the operations the end-user can perform in the target interface. When the designer performs an operation in this mode, Pavlov records 1) a stimulus-response pair mapping the physical action used to the operation that was executed, and 2) the first part of a second stimulus-response pair that will eventually map the operation to one or more operations demonstrated as the response.

Mapping the physical action to the operation is important because the drawing palette used during development to demonstrate operations does not appear when the target interface is executed (Test mode). In Test mode, the end-user can only use the operations that the designer has explicitly demonstrated as stimuli, and can only access those operations using the physical action (mouse button, auxiliary key) used in the demonstration. For example, in the driving simulator the end-user can only rotate the accelerator by dragging the mouse with the left-mouse button down, because that is how it was demonstrated. The designer cannot manipulate the car directly in any manner, because no such stimulus was demonstrated. This positive example method of specifying the functionality of the system is in contrast to the scheme of [9] in which the designer "freezes" the objects that cannot be manipulated.

The second recording made from a Stimulus demonstration records the high-level operation demonstrated as a stimulus (e.g., Accelerator.Rotate). It is the execution of this high-level stimulus that will trigger the execution of the responses demonstrated in Response mode.

The only complication to the semantics of a stimulus demonstration is that a designer may demonstrate a stimulus on a *representative* object. At run-time, the same stimulus applied to any member of the set represented will trigger the demonstrated response. Dynamically allocated objects, which can be specified by creating an object in stimulus or response mode (see [13]), are by default marked as representative objects. Pavlov also allows the designer to designate behavior groups, and marks each element as representative of the group (a similar approach is used in [12]).

**The Semantics of a Response Demonstration**
When the designer demonstrates an operation R on object O in response mode, the system connects a response of the form "O.R $(r_1,r_2,...)$ when C" to the previously demonstrated stimulus, where each $r_i$ is a function of zero or more stimulus parameters, and C is an optional context for when R should be executed.

The simplest semantic rule is to execute the demonstrated response each time the demonstrated stimulus occurs in the target interface. However, such a simple rule would preclude the designer from demonstrating the context for when an operation should be executed; semantics such as "execute R is response to stimulus S only when the environment is in state s" could not be demonstrated.

By setting a toggle, the Pavlov designer explicitly states if context should be taken into account. If it is, the designer configures the interface into the desired context (or the negation of the desired context) prior to a response demonstration. After the demonstration, the system runs a set of tests to identify graphical conditions describing the state of the interface. The designer is allowed to select one or more of these conditions and combine them with logical operators to define the context for when a response should be executed.

In the driving simulator, a context was defined on the description mapping the physical stimulus "Accelerator.LeftDrag" and the response "Accelerator.Rotate". A context could also be demonstrated so that the bottom car doesn't run into the top one: the designer demonstrates the stimulus of rotating the accelerator, tells the system to identify context, then demonstrates a response of moving the bottom car so that it intersects the top car. When the system identifies BottomCar.Intersects(TopCar) amongst other conditions, the designer selects it and negates it, and the following behavior is recorded:

```
(5) On Accelerator.Rotate(s1)->
      BottomCar.MoveForward(47.22*s1) At 0 Every 1
       When Not (BottomCar.Intersects(TopCar))
```

In general, there are many true conditions concerning the state of the interface. To reduce the number of conditions listed for the designer, the system only identifies those conditions relating the response object and all other objects in the interface. Because the stimulus object has also been signified as important, relationships found concerning it and the response object are shown at the top of the list of found conditions. When necessary, the designer can use the editor to specify a condition not identified by the system.

A second complication to the response semantics is similar to that discussed in the stimulus section: if the demonstrated response object is representative of a set, the response is applied to the entire set during execution (or a subset, as defined by a context conditional [13]).

A third complication to the response semantics concerns determining the response parameters. The simplest solution is, of course, to execute R during execution with the same parameters as in the demonstration. This is the best solution when the corresponding stimulus has no parameters (e.g., the stimulus is a button click and the response is a move(x=5,y=7)). However, for stimuli that do have parameters, it is often the case that the reaction is proportional, i.e., the response parameter(s) are proportional to the parameters of the stimulus. For instance, the car in the Driving Simulator is rotated an amount proportional to the amount the steering wheel is rotated. Thus, when such a stimulus-response is demonstrated, Pavlov infers proportional constants $C_i = r_i/s_i$ that relate each of the stimulus and response parameters. When the stimulus $S(s_1,s_2,...)$ occurs during execution, the response $R(s_1 \cdot C_1, s_2 \cdot C_2)$ is executed.

The formula for R illustrates that the system infers the first parameter of the stimulus to be related to the first parameter of the response, the second to the second, and so on. The basis of this inference is that most interface operations either have a single parameter or they have two parameters denoting x and y coordinates, so in practice the corresponding stimulus and response are often related. As with conditionals, the editor can be used to modify the response formulas recorded.

## EXTENSIONS FOR ANIMATION

Systems for demonstrating animation have existed for over twenty-five years [1]. Pavlov's contribution is the integration of animation demonstration with the stimulus-response model for defining interaction.

An animation path can be demonstrated with a real-time response demonstration, with a series of time-stamped response demonstrations (the editor can be used for in-betweening), or with a periodic response. Like any other response, an animation path can be triggered by any kind of end-user or time stimulus.

When a designer demonstrates the transformation of an object in real-time response mode, the system records es of time-stamped operations. Because operations are orded instead of picture frames (as in *Director*), the orded path is not constrained to a particular starting nt or object. Thus, reuse is facilitated. The mechanism also slightly more general than in systems such as *actor* because any operation, not just move, can be onstrated in real-time.

**Notion of Direction**

important contribution of Pavlov is that a designer can onstrate animation in which the end-user not only iates movement, but accelerates it and changes its ection. Though such behavior is the primary activity in ny game-like applications, there has been little research this area, and commercial systems such as *Director* quire extensive programming to develop this part of an plication.

m struggling with how to allow game-like behavior to demonstrated, the following observations were made: In ny games, one input control (e.g., steering wheel) is ed to control direction, and a different control celerator) to control speed. Also, many objects do not ve in an arbitrary manner, but are restricted to moving rward and backward, and must rotate their base to turn. om these observations it became clear that the standard ove(x,y) operation in Pavlov's drawing editor is not fficient for the demonstration of movement because it ecifies both a distance and an absolute direction.

solve the problem, a notion of direction was added to e stimulus-response model. Designers can set a *current ection* attribute for an object that is displayed during velopment. The direction attribute makes it possible for e designer to demonstrate a *MoveForward(d)* operation. is operation causes an object to move forward (or ckward, if d<0) in the direction it is facing. Thus, it is uch better suited for the demonstration of acceleration an Move(x,y).

**Periodic Responses**

The notion of a periodic response is also useful in demonstrating acceleration. In the driving simulator, when the end-user rotates the accelerator, the car should begin moving and *continue to move*, even after the end-user releases the mouse. In Pavlov, the designer explicitly specifies continuous movement by setting the *Every:* box on the clock before the demonstration of a forward movement. In essence, when a "MoveForward (d) Every t" is demonstrated, the system infers that the object should move forward at a speed of d/t.

The following run-time rule follows from these semantics: the execution of successive periodic MoveForward operations on the same object results not in two alternating and possibly opposite actions, but in a single action combining the magnitudes of the operations. For example, in the driving simulator, when the car is already moving at 4 units/frame and the end-user rotates the accelerator again, say back towards the origin, it causes a response of MoveForward (-1) units/frame. This second operation is combined with the existing one so that the car slows down to 4-1=3 units/frame; instead of alternating between moving forward 4 units, and backward 1 unit.

The system only uses these semantics for periodic MoveForward operations. For other operations, successive periodic responses will execute in tandem. Thus, using two periodic, regular Move demonstrations, the designer can demonstrate that an object move back and forth, such as in an *animated move icon.*

An alternative method of demonstrating acceleration has also been added to the Pavlov environment. After demonstrating a stimulus that should cause the acceleration, the designer enters *real-time* response mode and moves an object forward at the desired speed. Generally, a real-time response is used to demonstrate a fixed animation path as a response to a simple button-click or time. When a real-time Move Forward is demonstrated as a response to a transformational stimulus (one with parameters), the system does not record a series of discrete time-stamped operations as usual, but instead records a single periodic operation. The distance parameter is computed by dividing the total distance of the demonstrated movement by the time of the movement (d/t), and the period is set to 1 (ms).

The advantage of this scheme is that the designer truly demonstrates the speed of the movement; the disadvantage is it complicates the semantics of the system. A more thorough analysis will be provided after more feedback is gathered from users.

## THE PAVLOV EDITOR

An important aspect of a PBD system is how a designer edits the behaviors inferred by the system. Pavlov's editor borrows from *Director* by providing a time-line view of activity (a score). However, because interaction is emphasized, Pavlov provides multiple timelines: one for the events that occur without an end-user stimulus, and one for the events triggered by each end-user stimulus that was demonstrated. This method of organizing events by stimulus significantly eases the editing task compared to the single score editors found in most animation systems.

Pavlov's editor, shown in Figure 2, can be viewed simultaneously with the main development window. In order to view the operations that occur in response to a particular stimulus, the designer selects an object and a particular stimulus in the top-left list boxes. To view the operations that occur without an end-user stimulus, the designer selects the "Time Stimulus" check box to the right of the stimulus list.

The objects that respond to the listed stimulus are shown in the rows of the score. The designer can select a particular response in a cell, and the inferred response parameters appear in the edit boxes labeled R1 and R2. Any expression consisting of constants, stimuli parameters, and system-supplied object attribute functions may be entered as a response parameter. In essence, editing behavior formulas is very similar to entering a formula in a spreadsheet.

## IMPLEMENTATION

At the beginning of execution (Test mode), the time stimulated operations defined in the target interface are placed in the *execution list* with their respective time stamps. For each end-user stimulus that occurs, the *sr processor* traverses the selected object's stimulus-response list to find the response operations associated with the stimulus. These responses are copied into the execution list with a time-stamp of $t + t_r$, where t is the time the stimulus occurred (the current time), and $t_r$ is the recorded time-stamp of the response (which is relative to the stimulus). When the system is not processing end-user stimuli in this manner, the execution list is traversed and all operations whose time-stamp is less than the current time are executed. A non-periodic operation is removed from the execution list immediately after execution; a periodic operation is left in the list, with its time-stamp incremented by the size of its periodic interval. In either case, the executed operation is sent as a stimulus to the sr processor, so a chaining of events can occur.

Though the scheme does not guarantee that operations will be executed before or on their time-stamp, in practice it provides visually acceptable performance even for interfaces with lots of interaction and concurrent animation (Pavlov runs on a 486 PC).

## RELATED RESEARCH

*Rehearsal World* [5] and *Peridot* [7] were early PBD systems that inspired the stimulus-response framework. The first systems to allow direct graphical demonstration of a full range of stimuli and responses were *DEMO* [13] and *Marquise* [8]. *DEMO* introduced the stimulus-response model and a technique for demonstrating dynamically created objects, while *Marquise* focused on the demonstration of graphical editors, including those with palettes and modes.

*DEMO II* [3] and [4] are stimulus-response systems that allow the designer to perform multiple demonstrations of the same behavior to refine the system's inferences. [4] uses multiple examples to make sophisticated inferences concerning response parameters-- inferred formulas may depend on attributes of arbitrary objects as well as stimulus parameter values. *DEMO II* uses multiple examples to refine inferences concerning the context for when a response should be executed.

Pavlov is the first *stimulus-response* system to focus on animation, though there are a few PBD systems not based on stimulus-response that allow some animation to be demonstrated: *KidSim* [12] and *Agent Sheets* [10] use graphical rewrite rules to allow designers to demonstrate the context for when an operation should be executed. The systems are powerful for creating non-interactive simulations, but the rewrite-rule method of defining context is not integrated with a method of specifying end-user stimuli, so interactive simulations cannot be designed without coding; *Dance* [11] allows the demonstration of animation for the purpose of program visualization; *Chimera* [6] and *LEMMING* [9] allow interface behavior to be specified with multiple demonstrations or *constraints*, but do not cover time-based animation or acceleration.

*Director* is representative of the commercial animation systems that provide facilities for both animation and interaction design. These systems allow animation to be designed quickly and easily using a combination of frame-by-frame animation, in-betweening, and real-time recording. These systems also allow sound and video to be linked into presentations, and provide a range of features for creating special effects such as *slow-in/slow-out*, *motion blur*, and *squash and stretch*.

Though powerful for defining animation, these systems do not provide a PBD method of defining interaction. The systems all allow button-click triggered animation to be defined in a relatively simple manner. However, more complex stimulus-response behaviors, such as the steering wheel and accelerator controlled animation in the driving simulator, require expert-level programming.

A second difficulty in defining interaction with animation systems is that they are based on a single-score editor: all the animation sequences of an application are shown on a single time-line. Though such a score is sufficient for non-interactive animation (which was its original purpose), it is too unstructured for applications with interactive as well as time-stimulated animation. Like the programs written before the advent of structured programming (sub-procedures), the designer is forced to program control, i.e., where one animation ends and another begins, using *goto* statements. For complex applications with lots of movement and interaction, the result is a *spaghetti score*.

multiple score scheme in the Pavlov editor alleviates problem, and allows the designer to edit the different active behaviors and animation sequences separately.

## TATIONS

major practical limitation of Pavlov is that interfaces ed with it cannot be connected to application code. In next version, designers will be able to make this ection by 1) demonstrating a function call as a ulus or response, and 2) calling an application tion within a response or conditional formula. lov's single demonstration scheme might also be idered a limitation: more behaviors could be inferred multiple demonstration inference engine, such as [4], integrated. Before doing so, however, we want to study ther the additional inferred operations justify the itional complexity that would be added to the ironment. A third limitation is that *acceleration* can be onstrated for an object that has no pre-defined path, cannot be demonstrated for an object that must stay on *ed path*. In this regard, we are exploring both the use parametric functions to model some animation paths, the use of "conductor" objects with special properties ).

## MMARY

vlov contributes a cohesive model for demonstrating mation and interaction, and innovative techniques for onstrating interfaces in which the end-user controls h the speed and direction of animation paths. Using se techniques, interfaces like the driving simulator can created in less than fifteen minutes.

velopment is by no means restricted to driving ulators or similar applications. A number of other erfaces have also been developed, including a wide iety of games, a diagram editor with animated icons, d an educational solar system program. We attribute general usefulness of Pavlov to the generality of the mulus-response model, and its powerful multiple-score-sed editing facility.

Besides increasing the range of PBD, the stimulus-response model provides a very intuitive method for defining interfaces. A usability test was performed with a number of non-technical designers. Subjects were given a manual describing the stimulus-response model, and then were asked to design two interfaces equal in complexity to the driving simulator. Seven of ten were able to create the interfaces within an hour; the three others completed the tasks after asking a few questions (specific questions concerning how to complete the tasks were not allowed). We were extremely encouraged by the results, as well as the enthusiasm the subjects expressed for exploring once the formal tests were completed (though we could get none to salivate!).

## REFERENCES

1. Baecker, R., Picture-Driven Animation, *Proceedings of the Spring Joint Computer Conf.*, AFIPS Press, 1969, pp. 273-288.

2. Cypher, A., ed., *Watch What I Do: Programming By Demonstration*, MIT Press, Cambridge, Mass., 1993.

3. Fisher, G., Busse, D., and Wolber, D.,"Adding Rule Based Reasoning to a Demonstrational Interface Builder,*Proceedings of UIST'92*,Nov. 1992, pp.89-97.

4. Frank, M. and Foley, J., "A Pure Reasoning Engine for Programming By Demonstration", *Proceedings of UIST '94*, Nov. 1994, pp. 95-102.

5. Gould, L. and Finzer, W., "Programming By Rehearsal", *Byte*, v. 9, no. 6., 1984.

6. Kurlander, D. and Feiner, S., "Inferring Constraints from Multiple Snapshots", *ACM Transcations on Graphics*, May, 199[illegible].

7. Myers, B., *Creating User Interfaces By Demonstration*, Academic Press, San Diego, 1988.

8. Myers, B., McDaniel,R.,Kosbie, D.,"Marquise: Creating Complete User Interfaces By Demonstration, *Proceedings of INTERCHI '93*, Amsterdam, April,1993, pp.293-300.

9. Olsen, D., Ahlstrom. B., Kohlert, D., ,"Building Geometry- based Widgets by Example", *Proceedings of CHI '95*, May, 1995, pp.35-42.

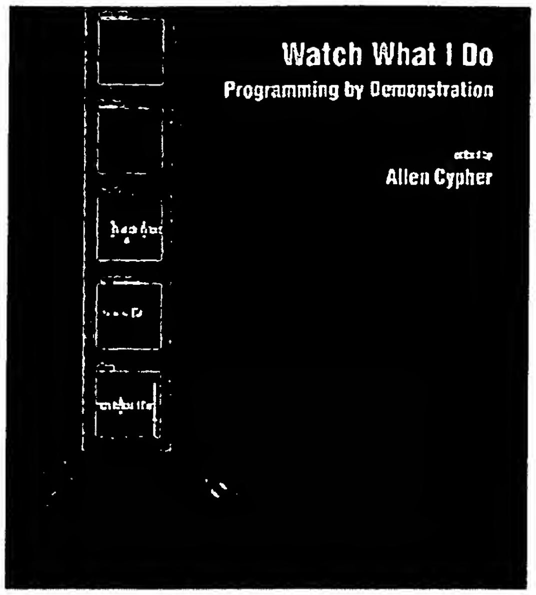
10. Repenning, A., "Agent Sheets: A Medium for Creating Domain-Oriented Visual Languages", *Computer*, V.28, 1995, pp. 17-25.

11. Stasko, J., "Using Direct Manipulation To Build Algorithm Animations By Demonstration", *Proceedings of CHI '91*, 1991, pp. 307-314.

12. Smith, D.C., Cypher A., "KidSim: End-User Programming of Simulations", *Proceedings of CHI '95*,May 1995, pp.27-34.

13. Wolber, D., and Fisher, Gene, "A Demonstrational Technique for Developing Interfaces with Dynamically Created Objects." *Proc. of UIST '91*, 1991, pp. 221-230.

The sequel to this book is Your Wish Is My Command



# Watch What I Do: Programming by Demonstration

edited by Allen Cypher

co-edited by Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky

1993

The MIT Press
Cambridge, Massachusetts
London, England

## Table of Contents

The **entire text** of this book is included on this web site. Access it through the Table of Contents.

## Book Reviews

- BYTE December 1993
- International Journal of Man-Machine Studies December 1993
- Sci Tech Book News December 1993
- GIS World February 1994
- SIGCHI Bulletin April 1994
- Burrelle's May 1994
- Journal of Visual Languages and Computing September 1994
- User Modeling and User-Adapted Interaction 1994

## See Also:

To purchase the book from the MIT Press, go to Watch What I Do
The sequel to this book is Your Wish Is My Command
Visit the home page for Programming by Example

# Contents

---

back to ... Watch What I Do ← Allen Cypher

Chapter 1

# Pygmalion: An Executable Electronic Blackboard

David Canfield Smith

## Introduction

*Pygmalion* was an early attempt to change the process of programming from one in which algorithms are described abstractly in a programming language to one in which they are demonstrated concretely to the machine. It introduced two new concepts: programming by demonstration and icons. Icons have become widely accepted. Programming by demonstration is still waiting for a practical system to show its validity. *Pygmalion* was completed in 1975. This chapter is a commentary on that work.

## Human-Computer Communication

Traditionally people instruct computers by sitting down with a pad of paper (or a window for a text editor) and writing a sequence of statements in some programming language. They then compile the statements, link the resulting machine code with other run-time routines, and attempt to execute it. There are almost always errors. At this point programmers enter the debug-edit-recompile loop. They track down bugs using whatever means are available, edit the source text of their programs, recompile it, relink it, and reexecute it. There are almost always more errors. In fact, fixing a bug often introduces new bugs. This debug-edit-recompile loop can consume the majority of time spent on a program. And no one claims anymore that all bugs can be removed from any large program.

Writing static language statements interspersed with compile-run-debug-edit periods is obviously a poor way to communicate. Suppose two humans tried to interact this way! Specifically, it is poor because it is:

**(a) Abstract**
The programmer must mentally construct a model of the state of the machine when the program will execute, and then write statements dealing with that imagined state. In practice, for a program of any complexity this is impossible. People cannot handle as much complexity as computers can. A typical symptom of this is boundary errors: an unanticipated value is used in an operation, resulting in an error. Dividing by zero is a classic example. We must find a way to allow programmers to work with concrete values without sacrificing generality.

**(b) Non-Interactive**
The debug-edit-recompile loop takes a lot of time per bug fix, particularly as programs get large. This low productivity has led programmers to seek faster machines and compilers. But that misses the point. Instead of speeding up the debug-edit-recompile loop, programmers should eliminate it! A few programming languages such as Lisp and Smalltalk are in fact interactive. One can type in an expression and immediately get it evaluated and see the result. Such languages are more productive than non-interactive languages.

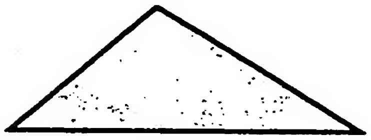
So why aren't interactive languages more widely used? Most are high level languages that are not as efficient as low level ones such as C and FORTRAN. In their search of a competitive advantage, software developers want their programs to execute as fast as possible, even at the expense of a longer development time. But this situation may be changing. Computers are now fast enough that interactive languages are practical in some cases. Several commercial applications have been written in Lisp, and IBM now recommends Smalltalk for developing applications under its OS/2 operating system. Even some versions of C now contain an interpreter for quick development and debugging.

**(c) "Fregean"**
The most articulate representation for a program requires the least translation between the internal representation in the mind and the external representation in

the medium. Aaron Sloman [Sloman 71] distinguishes two kinds of data representations: "analogical" and "Fregean" (after Gottlob Frege, the inventor of the predicate calculus). Analogical representations are similar in structure to the things they describe; Fregean representations have no such similarity. Consider the following examples:

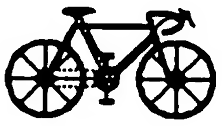
| |
|---|
| 1.23 |
| 45.98 |
| 17.1 |
| 63.709 |
| 125.4 |

array A



triangle

United States



bicycle

*Analogical*

*Fregean*

One of the advantages of analogical representations over Fregean ones is that structures and actions in a context using analogical representations (a "metaphorical" context) have a functional similarity to structures and actions in the context being modeled. One can discover relationships in a metaphorical context that are valid in the other context. For example, there are five items in array A; triangles have three sides; bicycles have two wheels; Texas is in the south. Analogical representations suggest operations to try, and it is likely that operations applied to analogical representations would be legal in the other context, and vice versa. This is the philosophical basis for the design of the Xerox Star and, ultimately, Apple Macintosh "desktop" user interfaces; they are a metaphor for the physical office [Smith 82]. Being able to put documents in folders in a physical office suggests that one ought to be able to put document icons in folder icons in the computer "desktop," and in fact one can.

Jerome Bruner, in his pioneering work on education [Bruner 66], identified three ways of thinking, or "mentalities":

- *enactive* - in which learning is accomplished by doing. A baby learns what a rattle is by shaking it. A child learns to ride a bicycle by riding one.
- *iconic* - in which learning and thinking utilize pictures. A child learns what a horse is by seeing one or a picture of one.
- *symbolic* - in which learning and thinking are by means of Fregean symbols. One of the main goals of education today is to teach people to think symbolically.

But Bruner argues that it is a mistake for schools to force children to abandon their enactive and iconic thinking skills when learning the symbolic. All three mentalities are valuable at different times and can often be combined to solve problems. All three skills should be preserved.

Jacques Hadamard, in a survey of mathematicians [Hadamard 45], found that many mathematicians and physicists think visually and reduce their thoughts to words only reluctantly to communicate with other people. Albert Einstein, for example, said that "The words of the language, as they are written or spoken, do not seem to play any role in my mechanism of thought. The psychical entities which seem to serve as elements in thought are certain signs and more or less clear images which can be 'voluntarily' reproduced and combined.... This combinatory play seems to be the essential feature in productive thought—before there is any connection with logical construction in words or other kinds of signs which can be communicated to others.... The above mentioned elements are, in my case, of visual and some of muscular type. Conventional words or other signs have to be sought for laboriously only in a secondary stage, when the mentioned associative play is sufficiently established and can be reproduced at will." [Hadamard 45, pp.142-3]

Some things are difficult to represent analogically, such as "yesterday" or "a variable-length array." But for concepts for which they are appropriate, analogical representations provide an intuitively natural paradigm for problem solving, especially for children, computer novices, and other ordinary people. And a convincing body of literature suggests that analogical representations, especially visual images, are a productive medium for creative thought [Arnheim 71].

Words are Fregean. Yet words and rather primitive data structures are often the only tools available to programmers for solving problems on computers. This leads to a translation gap between the programmer's mental model of a subject and what the computer can accept. *I believe that misunderstanding the value of analogical representations is the reason that almost all so-called visual programming languages, such as Prograph, fail to provide an improvement in expressivity over linear languages. Even in these visual languages, the representation of data is Fregean.*

With the advent of object-oriented programming, the data structures available have become semantically richer. Yet most objects are still Fregean. A programmer must still figure out how to map, say, a fish into instance variables and methods, which have nothing structurally to do with fish.

**(d) "Blank canvas" syndrome**

Pablo Picasso said "the most awful thing for a painter is the white canvas." One of his most memorable paintings, "The Studio at Cannes," is of his own studio. Its walls and ceiling are covered with paintings. Priceless works of art are jumbled together everywhere. But right in the middle of the room stands an easel holding a blank canvas. This is a fitting image of the problem facing all creative people: how to get started. A blank coding pad is as much a barrier to programming creativity as a blank canvas is to a painter.

What's needed is a lightweight, non-threatening medium like the back of a napkin, wherein one can sketch and play with ideas. The computer program described in the remainder of this chapter, *Pygmalion*, was an attempt to provide such a medium. *Pygmalion* was an attempt to allow people to use their enactive and iconic mentalities along with the symbolic in solving problems.

## Pygmalion

In Roman mythology, Pygmalion was a sculptor of extraordinary skill. At the peak of his powers, he determined to create a masterwork, a statue of a woman so perfect that it would live. The statue, which he named Galatea, was indeed beautiful, but alas it remained just stone. Heartbroken, he prayed to the gods. Venus, impressed with his work and passion, took pity on him and brought the statue to life. (They could do that back then.)

This urge to create something living is common among artists. Michelangelo is said to have struck with his mallet the knee of perhaps the most beautiful statue ever made, the Pieta, when it would not speak to him. And then there's the story of Frankenstein. Artists have consistently reported an exhilaration during the *act of creation*, followed by depression when the work is completed. "For it is then that the painter realizes that it is only a picture he is painting. Until then he had almost dared to hope that the picture might spring to life." (Lucien Freud, in [Gombrich 60], p.94) This is also the lure of programming, except that unlike other forms of art, computer programs *do* "come to life" in a sense.

The *Pygmalion* described in this chapter is a computer program that was designed to stimulate creative thinking in people [Smith 75]. Its design was based on the observation that for some people blackboards (or whiteboards these days) provide significant aid to communication. If you put two scientists together in a room, there had better be a blackboard in it or they will have trouble communicating. If there is one, they will immediately go to it and begin sketching ideas. Their sketches often contribute as much to the conversation as their words and gestures. Why can't people communicate with computers in the same way?

*Pygmalion* is a two-dimensional, visual programming environment implemented on an interactive computer with graphics display. (Although this work was completed nearly two decades ago, I will describe it in the present tense for readability.) It is both a programming language and a medium for experimenting with ideas. Communication between human and computer is by means of visual entities called "icons," subsuming the notions of variable, data structure, function and picture. Icons are sketched on the display screen. The heart of the system is an interactive "remembering" editor for icons, which both executes operations and records them for later reexecution. The display screen is viewed as a picture to be edited. Programming consists of creating a sequence of display images, the last of which contains the desired information. Display images are modified by graphical editing operations.

In the *Pygmalion* approach, a programmer sees and thinks about a program as a series of screen images or snapshots, like the frames of a movie. One starts with an image representing the initial state and transforms each image into the next by *editing* it to produce a new image. The programmer continues to edit until the desired picture appears. When one watches a program execute, it is similar to watching a movie. The difference is that, depending on the inputs, *Pygmalion* movies may change every time they are played. One feature that I didn't implement but wish I had is the ability to play movies both backward and forward; then one could step a program backward from a bug to find where it went wrong.

There are two key characteristics of the *Pygmalion* approach to programming.

- It relies on *editing* an artifact rather than typing statements in a programming language. Editing has proven to be easy for people. Everyone who uses computers—over 100 million people today—can use text and graphics editors, but hardly anyone can "program."
- The screen images always contain concrete examples of the program's data. This eliminates an entire class of errors due to abstraction.

Today we recognize these as being good user interface principles that most personal computer applications heed. So one way to characterize *Pygmalion* is to say that it attempted to bring good user interface principles to the *process* of programming, not just to the *result* of programming.

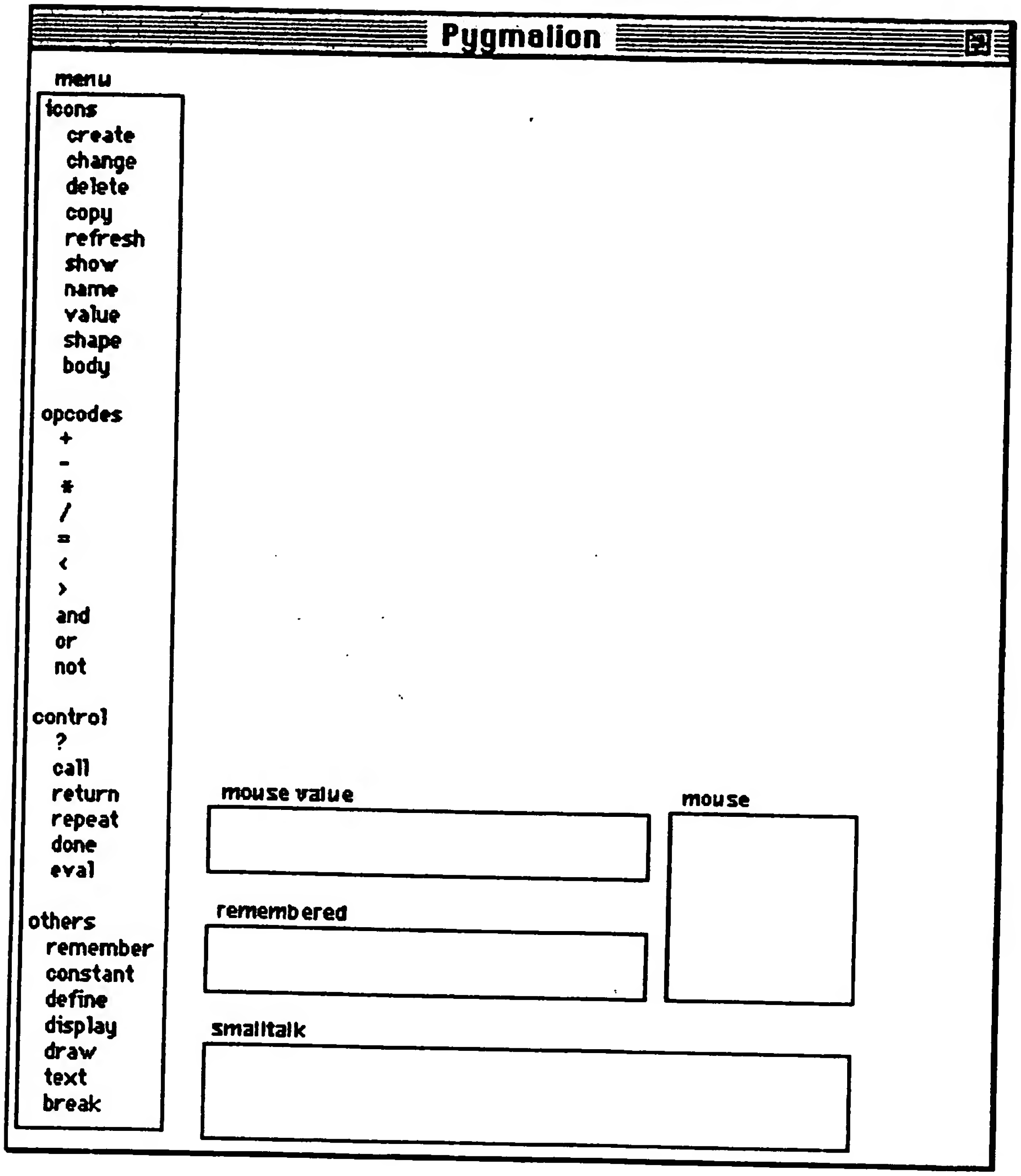
Another early attempt to specify programming by editing was Gael Curry's *Programming by Abstract Demonstration* [Curry 78]. It was similar to Pygmalion, except that instead of dealing with concrete values, Curry manipulated abstract symbolic representations (e.g. "n") for data. While this made some things easier to program than in *Pygmalion*, it is interesting to note that in his thesis he reported making exactly the kinds of boundary errors mentioned above because he could not see actual values.

## An Example of Programming by Demonstration: Defining Factorial

We will illustrate the *Pygmalion* way of programming by defining the function "factorial." An ALGOL-like definition might be:

```
integer function factorial (integer n) =
      if n = 1 then 1
      else n * factorial(n - 1);
```

In *Pygmalion*, a person would define factorial by picking some number, say "6", and then working out the answer for factorial(6) using the display screen as a "blackboard." In the illustrations below, in order to save space I won't show the entire display screen for each "movie frame." Rather, I'll often "zoom in" on the parts of the screen that change from frame to frame. The programmer, however, always sees the entire screen:


Pygmalion
menu
icons
create
change
delete
copy
refresh
show
name
value
shape
body
opcodes
+
-
*
/
=
<
>
and
or
not
control
?
call
return
repeat
done
eval
others
remember
constant
define
display
draw
text
break
mouse value
mouse
remembered
smalltalk

(Aside: the screen shots in this chapter are from a recent HyperCard simulation of *Pygmalion* done on a Macintosh computer by Allen Cypher and myself. Therefore the window appearance differs slightly from the actual system.)

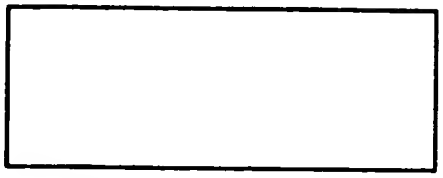
The *Pygmalion* window consists of several parts:

- "menu" area – a menu of icons and operations on icons (a rather mixed bag actually—today the icons would be in a palette and the operations in pull-down menus). The icons are under the "opcodes" and "control" headings, and the operations are under the "icons" and "others" headings. This is the complete icon editor.
- "mouse value" area – Programmers could type a value here, and it would become "attached" to the mouse. It could then be deposited in icons.
- "remembered" area – When the system is "remembering," the last couple of operations performed are displayed here.
- "smalltalk" area – Smalltalk expressions could be typed and evaluated here.
- "mouse" area – *Pygmalion* was implemented on a computer having a three button mouse; this area displayed the meaning of each button in the different modes.

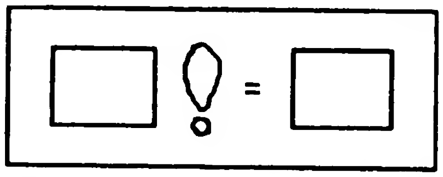
Except for the menu, these parts are not really important. As I mention at the end, I would get rid of them all were I to reimplement the system today. But with these as our resources, let's define factorial.

The entities with which one programs in *Pygmalion* are what I call "icons." An icon is a graphic entity that has meaning both as a visual image and as a machine object. Icons control the execution of computer programs, because they have code and data associated with them, as well as their images on the screen. This distinguishes icons from, say, lines and rectangles in a drawing program, which have no such semantics. *Pygmalion* is the origin of the concept of icons as it now appears in graphical user interfaces on personal computers. After completing my thesis, I joined Xerox's "Star" computer project. The first thing I did was recast the programmer-oriented icons of *Pygmalion* into office-oriented ones representing documents, folders, file cabinets, mail boxes, telephones, wastebaskets, etc. [Smith 82]. These icons have both data (e.g. document icons contain text) and behavior (e.g. when a document icon is dropped on a folder icon, the folder stores the document in the file system). This idea has subsequently been adopted by the entire personal computer and workstation industry.

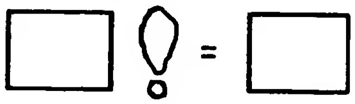
To get started, I define an icon for the function:



I create two sub-icons to hold the argument and value, and then draw some (crude) graphics to indicate the icon's purpose:



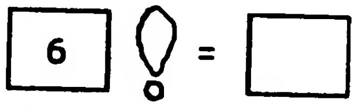
Finally I make the outside border invisible:



I now select the icon, invoke the "define" operation, and type in the name "factorial." This associates a name with the icon. The first thing the system does with a newly created function is to capture the screen state. Whenever the function is invoked, it restores the screen to this state. This is so that the function will execute the same way regardless of what is on the screen when it is called. So before invoking "define," I make sure that I clear off any extraneous icons lying around. Icons may be deliberately left on the screen when a function is defined; these act as global variables. When the function is invoked, the "global" icons and their current values are restored to the screen (if they weren't already present) and are therefore available to the function.

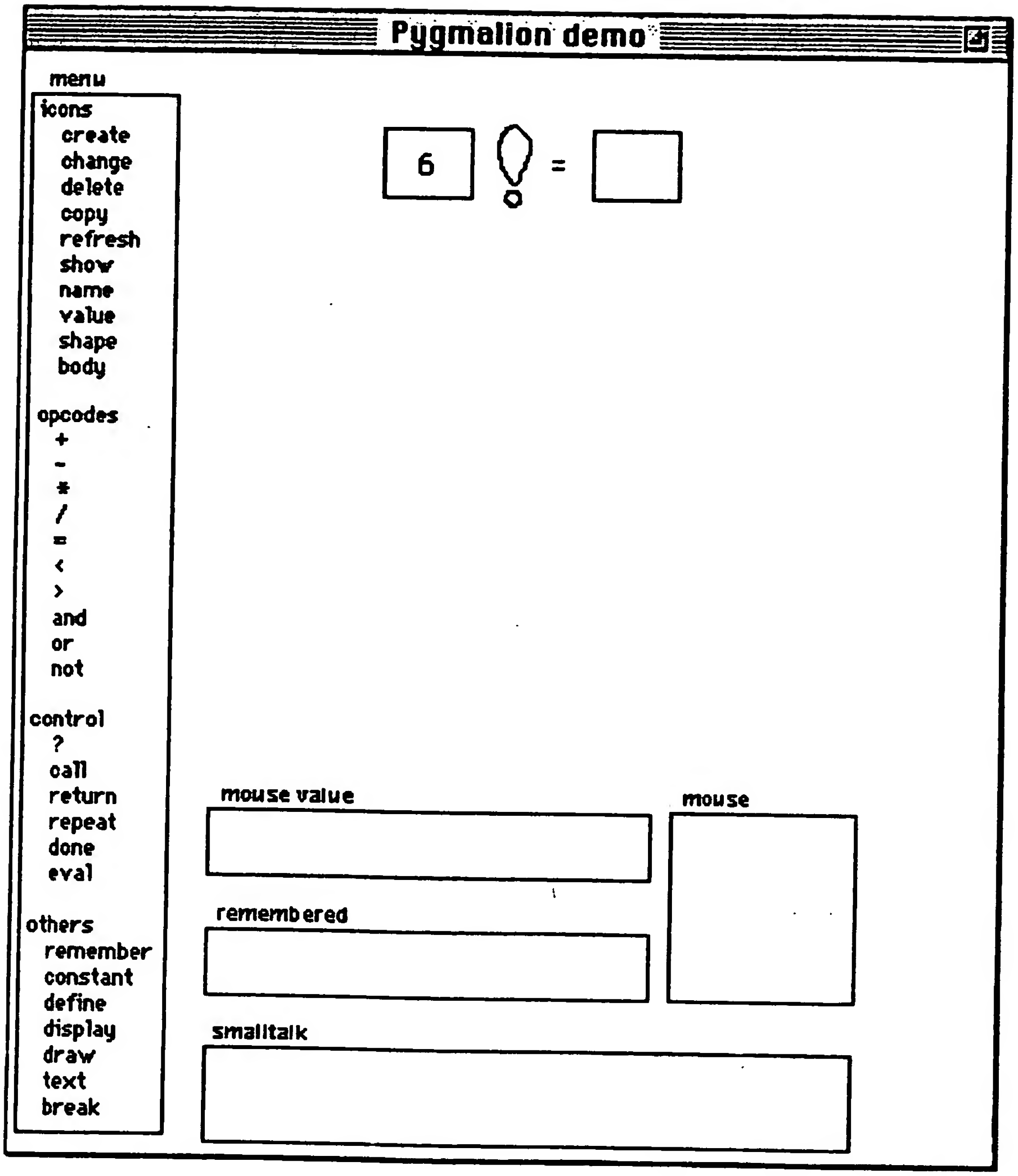
As soon as I invoke the "define" operation, the system enters "remember mode," causing every action I subsequently do to be recorded in a script attached to the icon. So one defines functions simply by *working out examples on the screen.* Programs are created as a side effect. Here I'll arbitrarily work out the value of factorial for the number "6". (Choosing good examples is an art. One wants both typical values and boundary cases. Even in conventional programming, choosing test cases poorly is a principal sources of bugs. *Pygmalion* does not solve this problem.)

I type "6" into the argument icon:



Whenever all the arguments to a function are filled in, the system immediately invokes it. This was an experiment and is not inherent in programming by demonstration, or even necessarily a good idea. Functions in *Pygmalion* can be invoked even if their code is undefined or incomplete. When the system reaches a part of the function that has not yet been defined, it traps to the user asking what to do next. This was implemented by placing a "trap" operator at the end of every code script. When a trap operator is executed, the system reenters "record mode." Every action subsequently performed is inserted in the code script in front of the trap. When the user invokes the "done" operation indicating that this code script is finished, the system leaves "record mode" and removes the trap operator. Otherwise the trap operator remains there.

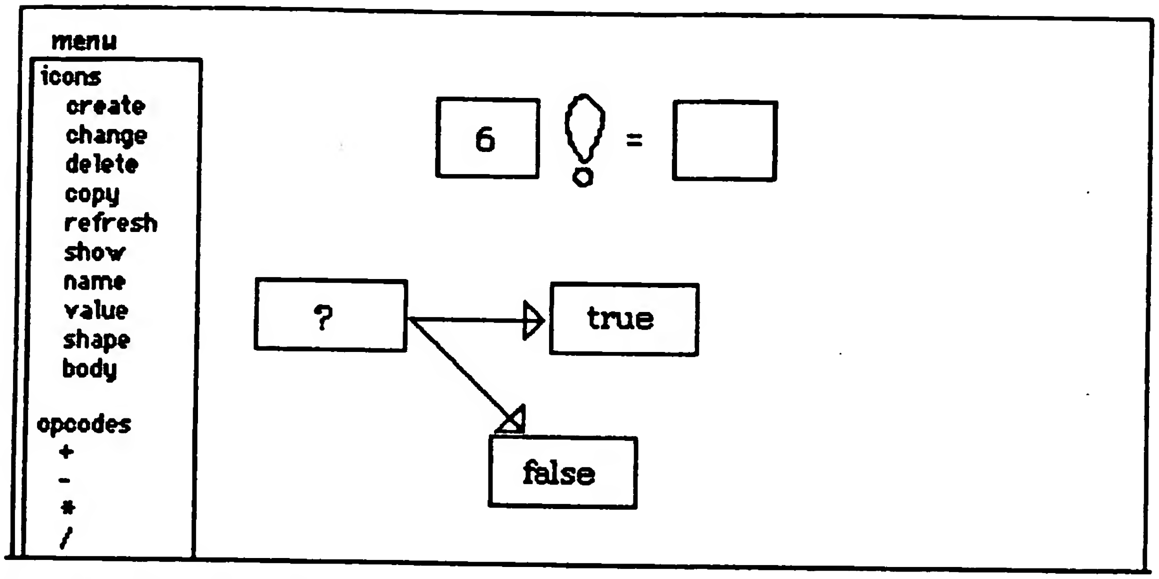
Factorial is now ready to execute, so the system invokes it. Since there is no code defined for it yet, it immediately traps to the programmer asking what to do. I move the factorial icon out of the way to make some room on the screen:


Pygmalion demo
menu
icons
create
change
delete
copy
refresh
show
name
value
shape
body
opcodes
+
-
*
/
=
<
>
and
or
not
control
?
call
return
repeat
done
eval
others
remember
constant
define
display
draw
text
break
6
=
mouse value
mouse
remembered
smalltalk

From here on, in order to save space in these pictures I won't show the window title or any area that is not directly involved in the actions being described.

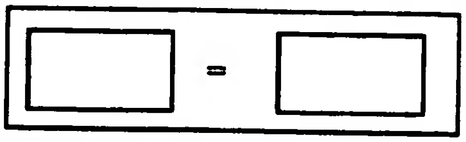
I want to test if the argument is equal to 1. Actually, I can see that it isn't 1; it's 6. So why do I have to make a test? The answer is that *Pygmalion is designed for programmers*. Programmers know that functions will be called with different arguments, and they try to anticipate and handle those arguments appropriately. This is a little schizophrenic of *Pygmalion*: on the one hand it attempts to make programming accessible to a wider class of users; on the other, it relies on the kind of planning which only experienced programmers are good at. But anticipation is not inherent in programming by demonstration. Henry Lieberman's *Tinker* [Lieberman 80, 81, 82] doesn't force users to anticipate future arguments. In *Tinker* one writes code dealing only with the case at hand. When *Tinker* detects an argument that is not currently handled, it asks the programmer for a predicate to distinguish the new case from previous ones. The programmer then writes code to handle the new case. *Tinker* synthesizes a conditional expression from the predicate, the new code, and the existing code. Programmers never have to anticipate; they need only react to the current situation. This is an improvement over *Pygmalion's* approach.

At any rate, I've decided that I will need a conditional, so I invoke the "?" item in the menu. The system enters a mode waiting for me to specify where to put the conditional icon. When I click the mouse, the icon is placed at the cursor location:

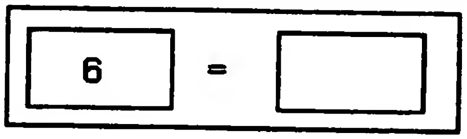


The conditional icon consists of three sub-icons: one for the predicate and two to hold the code for the true and false branches. The predicate icon is the "argument" to the conditional; as soon as it has a value, the appropriate branch icon executes.

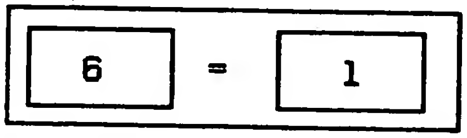
To test if the factorial argument is equal to 1, I click on the "=" menu item, point where I want the icon to go, and the system creates an equality-testing icon:



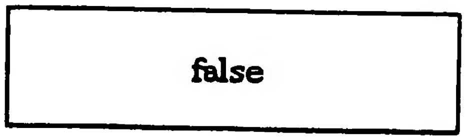
It has two sub-icons to hold the data being tested. I drag the "6" down from factorial's argument icon and deposit it in the left hand sub-icon:



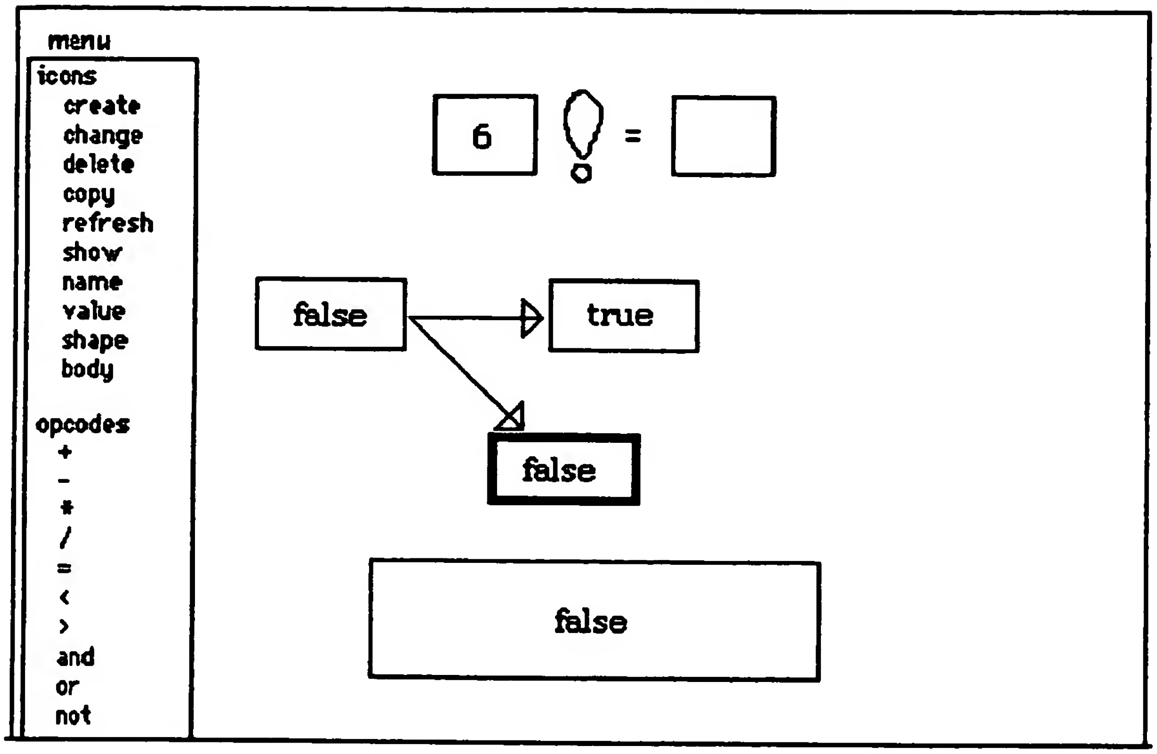
(During playback—"watching the movie,"—Curry's system animated such value dragging, making for a quite articulate movie; this was an improvement over *Pygmalion*, which did no such animation.) Then I type a "1" into the right hand sub-icon:



The equality icon is now fully instantiated, so it executes. It is defined to replace its contents with the result of the test. Since 6 is not equal to 1, it replaces its contents with the symbol "false":



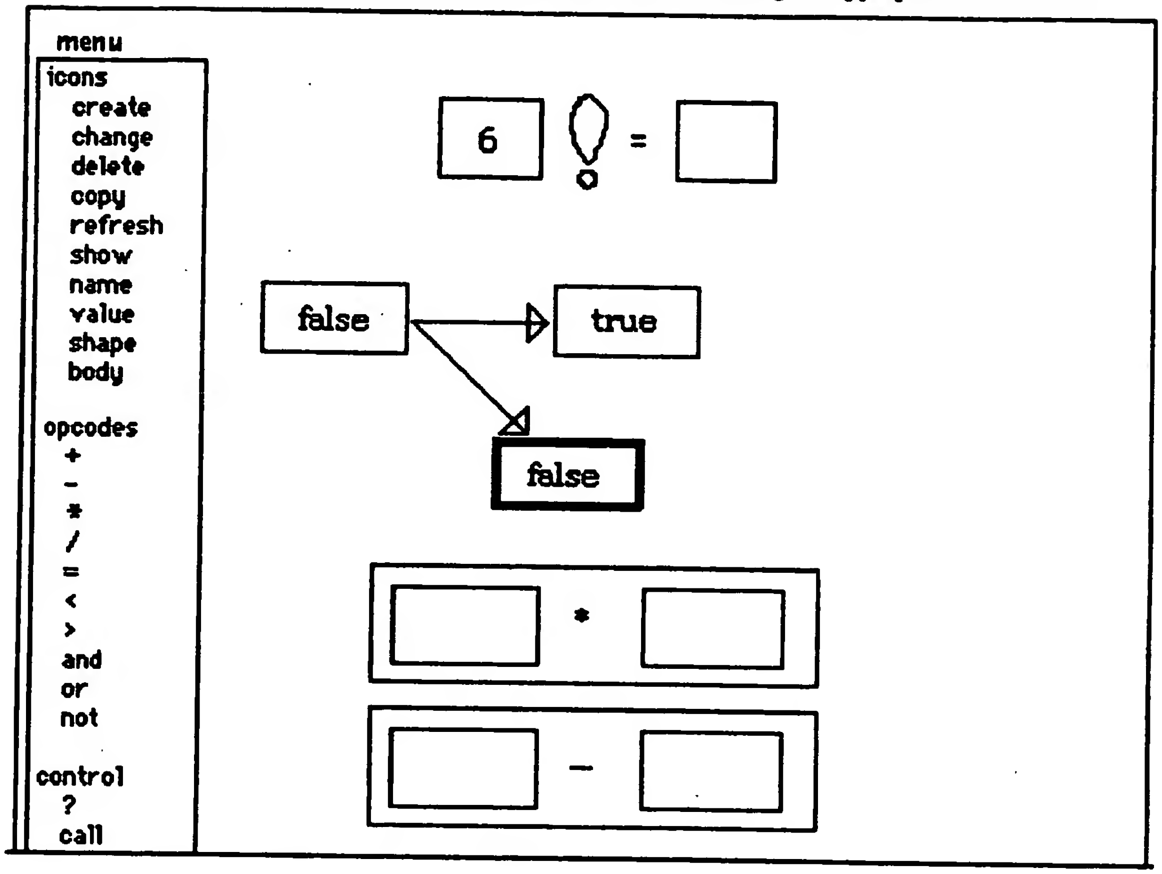
This value, like the value in any icon on the screen, can be used in other parts of the computation. In particular, it can be used in the conditional icon. I drag it into the conditional's predicate icon. The conditional icon is now fully instantiated, so it executes. The symbol "false" causes the false icon to be evaluated:



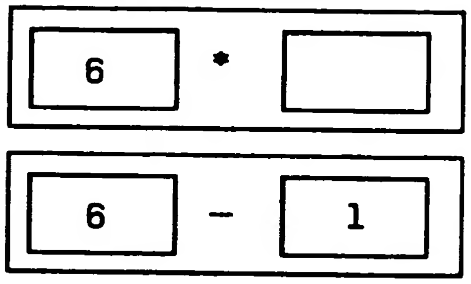
But the false icon has no code defined for it yet, so it immediately traps back to me asking what to do. The false icon now becomes the "remembering" icon; previously it was the icon for factorial itself. So now I'm no longer defining code for the factorial icon; I'm defining code for the false branch of its conditional icon. In fact, from here on out all of the operations I perform will be attached to either the true or false branch of the conditional.

Since I don't need the equality testing icon anymore, I'll get rid of it. I do this by invoking the "delete" menu item and then clicking on the equality icon.

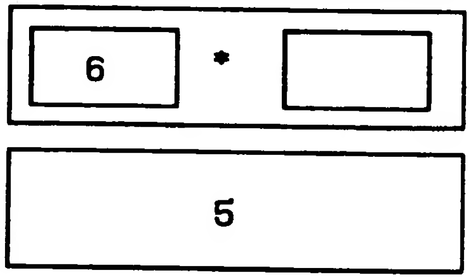
Now I want to compute `n*factorial(n-1)`. I'll need a multiplication icon and a subtraction icon. These I get by invoking the appropriate menu items:



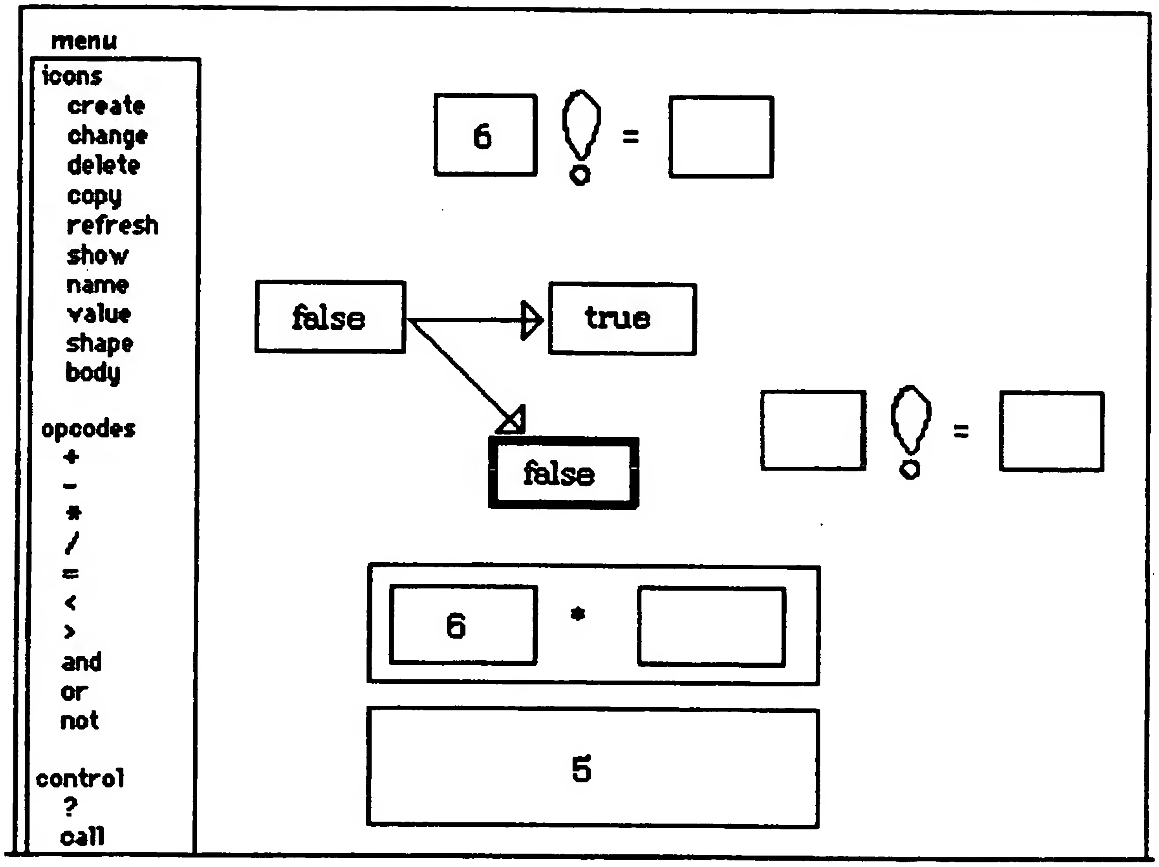
I drag factorial's argument (6) into the left hand icon in each of these. Then I type a 1 into the right hand icon in the subtracter:



The subtraction icon is now fully instantiated, so it executes:

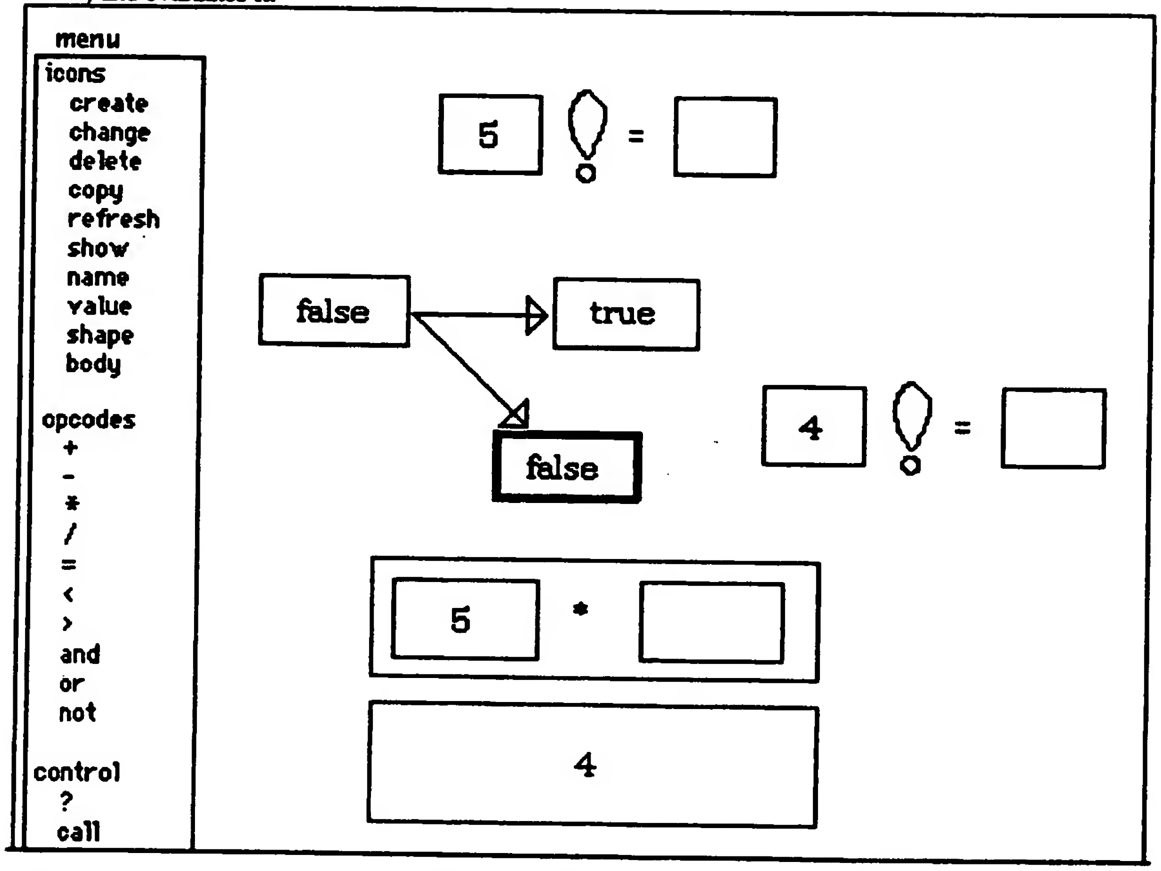


Now I want to call factorial recursively on this value (5). I invoke the "call" menu item, type "factorial" when prompted, and click where I want it to go. The system creates a new instance of the factorial icon, the very one that I am in the middle of defining! As mentioned earlier, that factorial is not completely defined poses no problems for *Pygmalion*; it will execute what it has and ask for more when it runs out. The screen now looks like this:



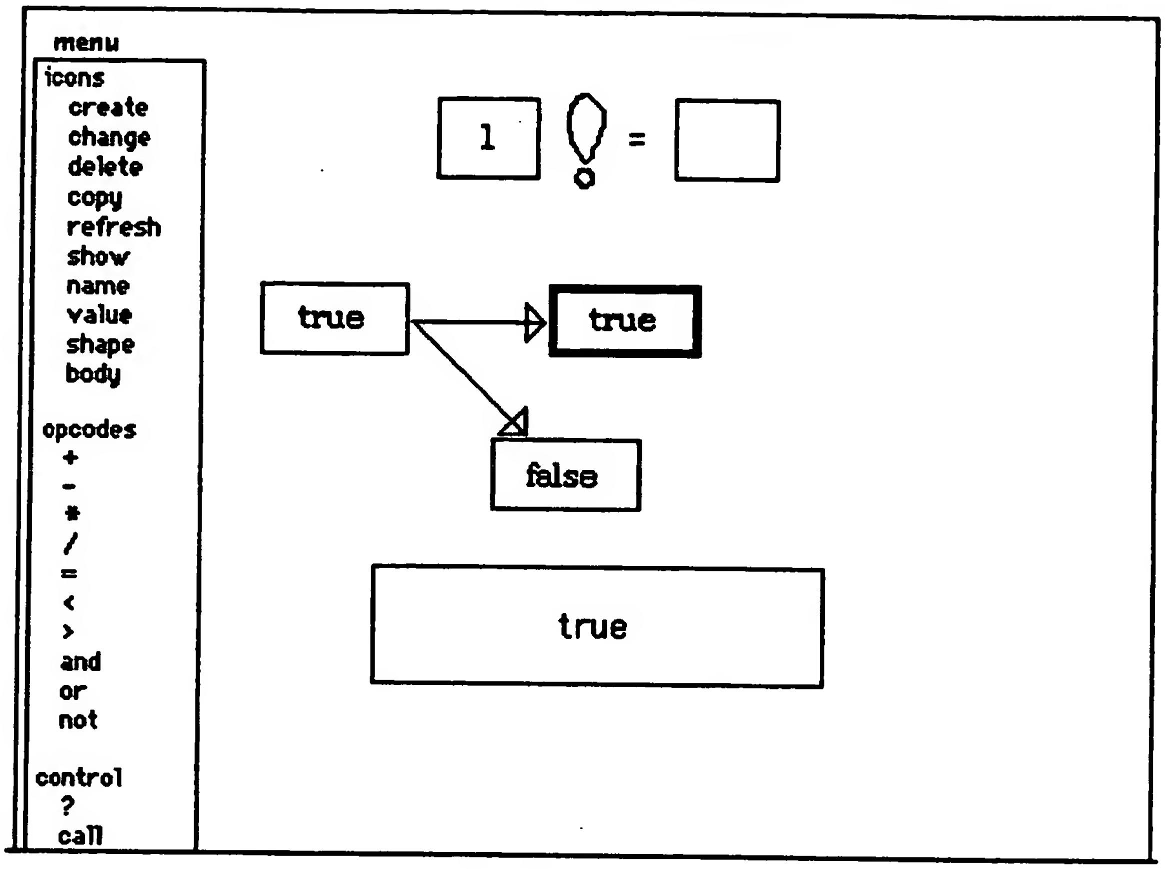
I drag the 5 into factorial's argument icon. As usual since it is now fully instantiated, it executes. But now instead of being empty, there is quite a bit of code for factorial to execute:

- It cleans off the screen and repositions the factorial icon.
- It creates a conditional icon.
- It creates an equality testing icon, fills it in, and evaluates it. Since 5 ≠ 1, it evaluates to false again.
- This causes the false branch of the conditional to be executed again.
- This creates multiplication and subtraction icons, fills in the subtraction icon, and evaluates it, producing 4.
- It creates yet another instance of the factorial icon, fills in the argument with 4, and evaluates it.

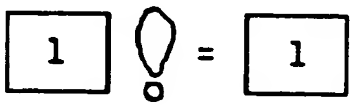


*The screen just before factorial(4) is evaluated*

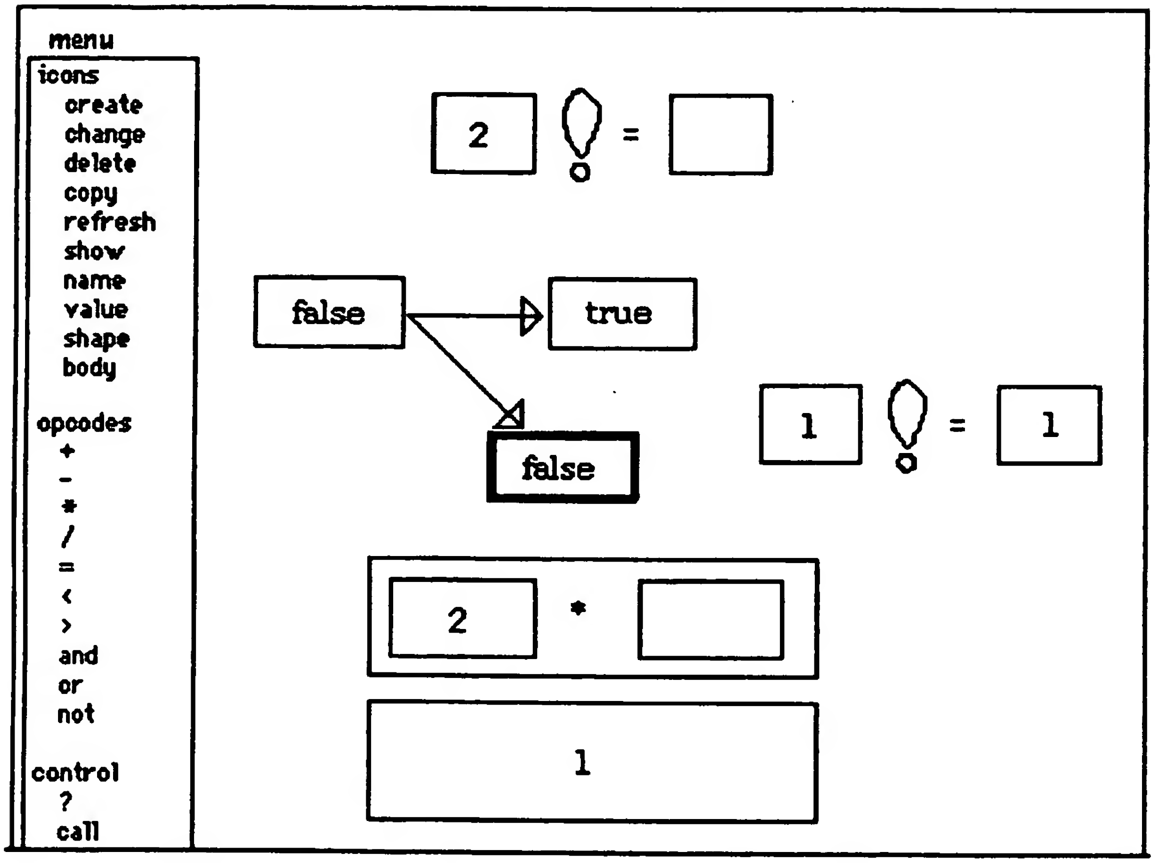
And so on recursively, until finally factorial is called with 1 as the argument. At this point, for the first time, the conditional's true icon is executed. But there is no code yet for this icon, so it immediately traps back to me asking for instructions. The screen now looks like this:



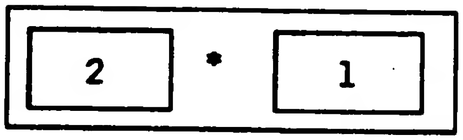
In this case, we know exactly what to do. Factorial(1) = 1. So I type 1 into the value icon:



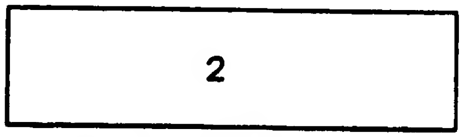
and invoke the "done" menu item. This returns from the last recursive call—factorial(1). It immediately traps asking for more instructions for the false branch of the conditional, since it was still trying to compute factorial(2) when the recursive call to factorial(1) was made. The screen now looks like this:



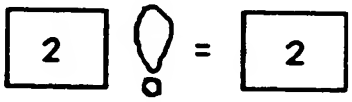
All that remains is to drag the value from the recursive call factorial(1) into the multiplication icon:



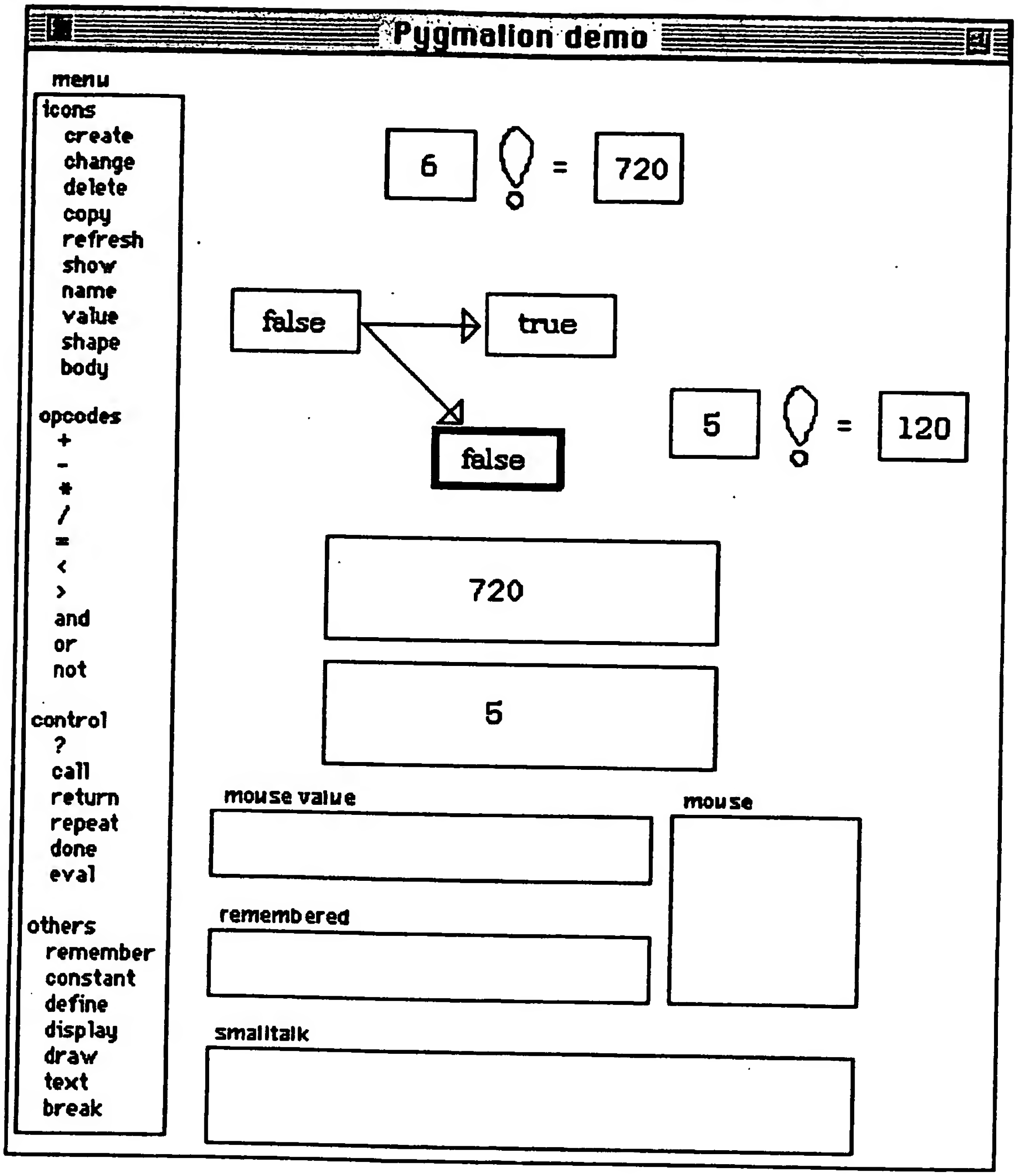
which causes it to evaluate:

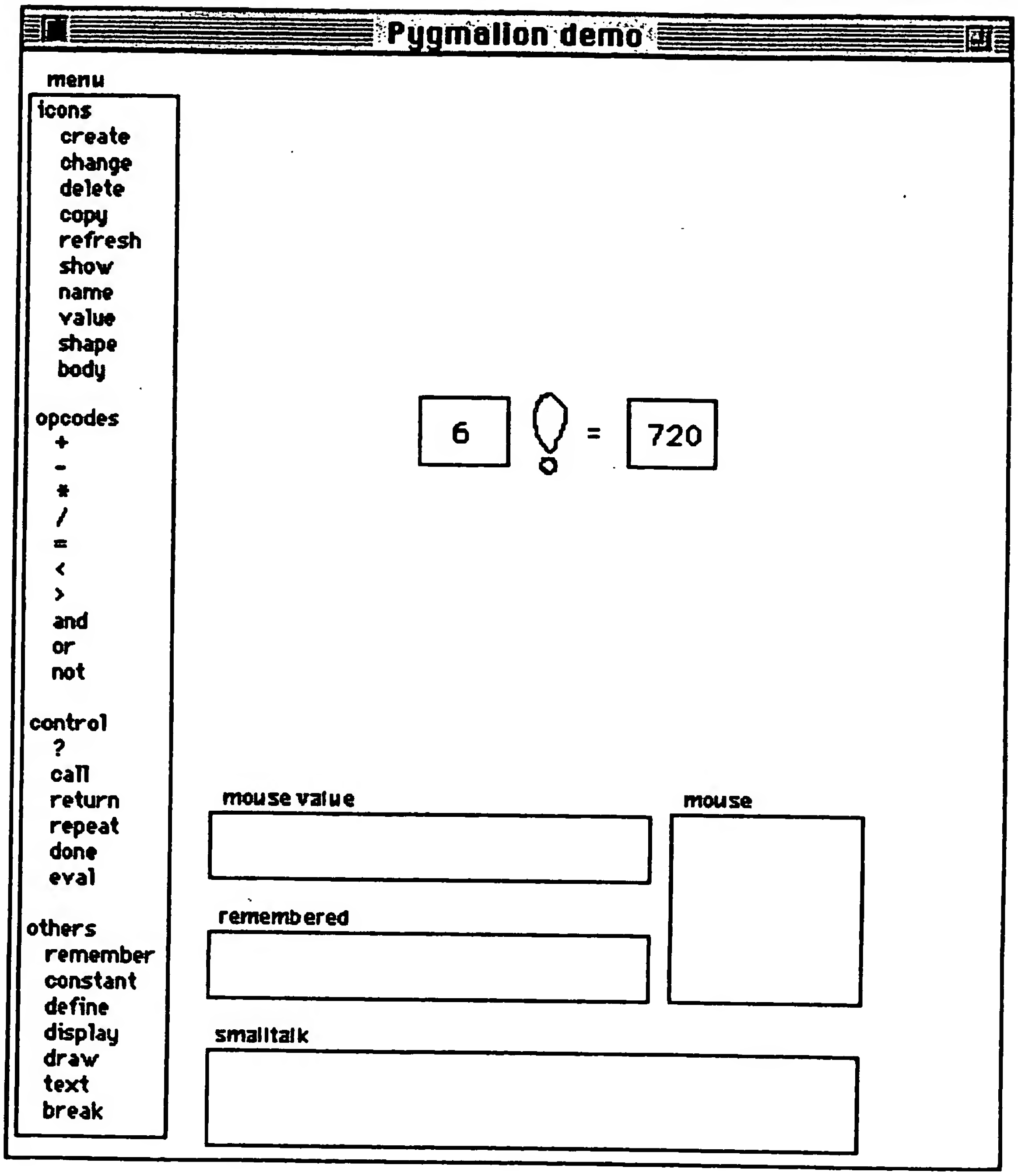


This then is `n*factorial(n-1)`, where n = 2. I drag this value into factorial's value icon—the false branch as well as the true branch must compute a value—and invoke "done."



The recursion now completely unwinds, since it knows how to do everything it needs to, and ends with the value of factorial(6). See the next page. Finally, in the picture on the following page, it restores the screen to its state when the function was initially invoked.


Pygmalion demo
menu
icons
create
change
delete
copy
refresh
show
name
value
shape
body
opcodes
+
-
*
/
=
<
>
and
or
not
control
?
call
return
repeat
done
eval
others
remember
constant
define
display
draw
text
break
6
=
720
false
true
false
5
=
120
720
5
mouse value
mouse
remembered
smalltalk


Pygmalion demo
menu
icons
create
change
delete
copy
refresh
show
name
value
shape
body
opcodes
+
-
*
/
=
<
>
and
or
not
control
?
call
return
repeat
done
eval
others
remember
constant
define
display
draw
text
break
6
=
720
mouse value
mouse
remembered
smalltalk

People sometimes ask me, if I were to build *Pygmalion* today, how would it be different? There are three main things I would do differently:

1. The most important change I'd make is to address a broader class of users. *Pygmalion* was designed for a specific target audience: computer scientists. These people already know how to program, and they understand programming concepts such as variables and iteration. I would like to see if the approach could be applied to a wider audience: business people, homemakers, teachers, children, the average "man on the street." This requires using objects and actions in the conceptual space of these users: instead of variables and arrays, use documents and folders, or cars and trucks, or dolls and doll houses, or food and spices. Or better yet, let users define their own objects and actions. Dan Halbert captured this idea nicely with his concept of "programming in the language of the user interface" [Halbert 84].

2. The biggest weakness of *Pygmalion*, and of all the programming by demonstration systems that have followed it to date, is that it is a toy system. Only simple algorithms could be programmed. Because of memory limitations, I could execute factorial(3) but factorial(4) ran out of memory. (*Pygmalion* was implemented on a 64 K byte computer with no virtual memory.) There was no way to write a compiler in it or a chess playing program or an accounting program, nor is it clear that its approach would even work for such large tasks. (I did, however, implement part of an operating system in it, at least on paper.) The biggest challenge for programming by demonstration efforts is to build a practical system in which nontrivial programs can be written.

3. I would put a greater emphasis on the user interface. Given what we know about graphical user interfaces today, it wouldn't be hard to improve the interface dramatically. I would put the icons into a floating palette as in HyperCard, replacing the long menu area that takes up so much screen space. I would put the operations on icons into pull-down menus. I would get rid of the bottom four areas in the *Pygmalion* window entirely. The "mouse" area was necessary because the system was pretty modal; I would redesign it to be modeless. I would make greater use of overlapping windows (which hadn't been invented when this work was done). I would improve the graphics esthetics. With today's graphics resources—bit mapped screens, processor power, graphics editors, experienced bit map graphics artists—there is no longer any excuse for poor appearance. Good

esthetics are an important factor in the users' enjoyment of a system, and enjoyment is crucial to creativity.

## Summary

*Pygmalion* was an early attempt to improve the process of programming. By studying how people think and communicate, I attempted to build a programming environment that facilitated communication and stimulated people's ability to think creatively. While it never went beyond a toy system, *Pygmalion* embodied some ideas that still seem to have promise today:

- It allowed ideas to be worked out via sketches on the screen and then was able to reexecute the sketches on new data.
- It introduced icons as the basic entity for representing and controlling programs.
- Programming was done by editing sketches and then recording the editing actions. This avoided the abstract step of writing down statements in a programming language.
- Example data were always concrete, never abstract.
- It used analogical representations for data. This reduced the translation distance between mental models and computer models of data.
- It represented programs as movies. (Storyboards would probably be better, à la David Kurlander [Kurlander 88b].)

To make it possible for the average person to write computer programs, I still believe that some combination of these ideas must be present. This is a worthy challenge for the 1990's.

## Pygmalion

Uses and Users

**Application domain:** Custom graphical programming environment
**Intended users:** Programmers

User Interaction

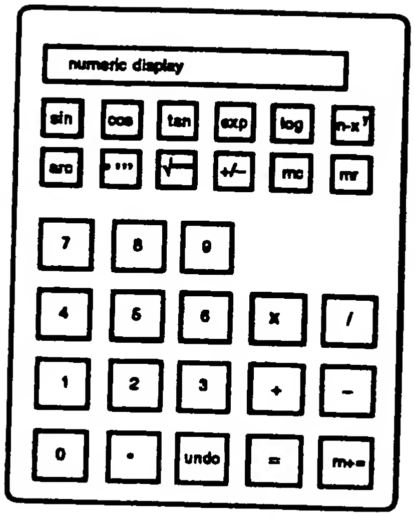
**How does the user create, execute and modify programs?**
The user creates programs by editing graphical snapshots of the computation. Essentially the user treats the display screen as an "electronic blackboard", using it to work out algorithms on specific concrete examples.
Pygmalion provides graphical representations for the standard arithmetic, relational, and Boolean operators.
Partially specified programs could be executed. The system asks the user what to do next when it reaches the end of a branch.
Programs can be modified, but then the rest of the modified branch must be re-demonstrated.

Inference

**Inferencing:** No inferencing
**Types of examples:** Multiple examples to demonstrate branches of a conditional, and recursion.
**Program constructs:** Variables, loops, conditionals, recursion

Implementation

**Machine, language, size, date:** Smalltalk, 1975.



| | |
|---|---|
| mc | clear memory |
| mr | retrieve from memory |
| m+= | add to memory |

# A Predictive Calculator

Chapter 3

Ian H. Witten

## Introduction

One way of giving casual users access to some of the power of a computer without having to learn formal programming methods is to enable tasks to be defined by giving examples of their execution, rather than by a procedural specification. This chapter summarizes a 1981 proposal for an electronic calculator that allows iterative computations to be inferred interactively from an initial part of the sequence of key-presses [Witten 81]. The aim was to construct a device that is partially self-programming. The predictive calculator was implemented on a PDP-11/40 computer in 1981. It was intended as a demonstration rather than a practical tool for casual users, and the user interface was not completely polished.

The idea of defining problems operationally through examples was proposed in the "query-by-example" system for database retrieval [Zloof 77b], which was intended for a user with no programming and little mathematical experience. To specify what to retrieve, you present an example of an item that should be included. The scheme illustrates that useful interactive systems can be built that allow users to define non-trivial tasks by example. One of its advantages is that it frees users from thinking of the retrieval problem in an artificially sequential form: instead they can specify the links, conditions, and constraints in any order.

*Figure 1. Trace of a sorting process*

```
t:=a(0) s:=0 i:=1 i<t-1 a(i+1)<a(i) s:=i x:=a(i)
a(i):=a(i+1) a(i+1):=x i:=i+1 i<t-1 a(i+1)>a(i) i:=i+1
i<t-1 a(i+1)<a(i) s:=i x:=a(i) a(i):=a(i+1) a(i+1):=x
i:=i+1 i<t-1 a(i+1)<a(i) s:=i x:=a(i) a(i):=a(i+1)
a(i+1):=x i:=i+1 i=t-1 s>0 s:=0 i:=1 i<t-1 a(i+1)>a(i)
i:=i+1 i<t-1 a(i+1)>a(i) i:=i+1 i<t-1 a(i+1)<a(i) s:=i
x:=a(i) a(i):=a(i+1) a(i+1):=x i:=i+1 i<t-1
a(i+1)>a(i) i:=i+1 i=t-1 s>0 s:=0 i:=1 i<t-1
a(i+1)>a(i) i:=i+1 i<t-1 a(i+1)<a(i) s:=i x:=a(i)
a(i):=a(i+1) a(i+1):=x i:=i+1 i<t 1 a(i+1)>a(i)
i:=i+1 i<t-1 a(i+1)>a(i) i:=i+1 i=t-1 s>0 s:=0 i:=1
i<t-1 a(i+1)>a(i) i:=i+1 i<t-1 a(i+1)>a(i) i:=i+1
i<t-1 a(i+1)>a(i) i:=i+1 i<t-1 a(i+1)>a(i) i:=i+1
i=t-1 s=0 end
```

| | |
|---|---|
| Input: `a(0)`. | number of elements to be sorted, minus 1 |
| `a(1) ... a(t-1)` | elements to be sorted into increasing numerical order |

In contrast, the present work explores how strictly *sequential* information can be inferred by a machine from an initial subsequence.

Automatic inference of programs from traces is another domain in which the problem is specified by an example. When a detailed trace of a particular execution of a program is available at the right level of generality, program inference can be easy. For instance, [Biermann 72] discusses the inference of Turing machines from traces of sample computations. The problem is more usefully addressed at a higher level than Turing-machine instructions, and [Gaines 76] gave the programming-language-level trace in Figure 1 as an example of a sorting program that performs a simple bubble-sort. The flowchart in Figure 2 can be derived from the trace simply by assigning different states to different statements and drawing sequential links between them. It is correct and complete except for a link from the assignment "`i:=1`" to the test "`i=t-1`" which was not demonstrated by that particular example. However, it is unlikely that such a trace would be entered without error, and even if it were, it may not be any easier for a casual user to generate than a structural description—a program—for the sorting process. Traces can be useful, though, in more limited domains—as this chapter will show.

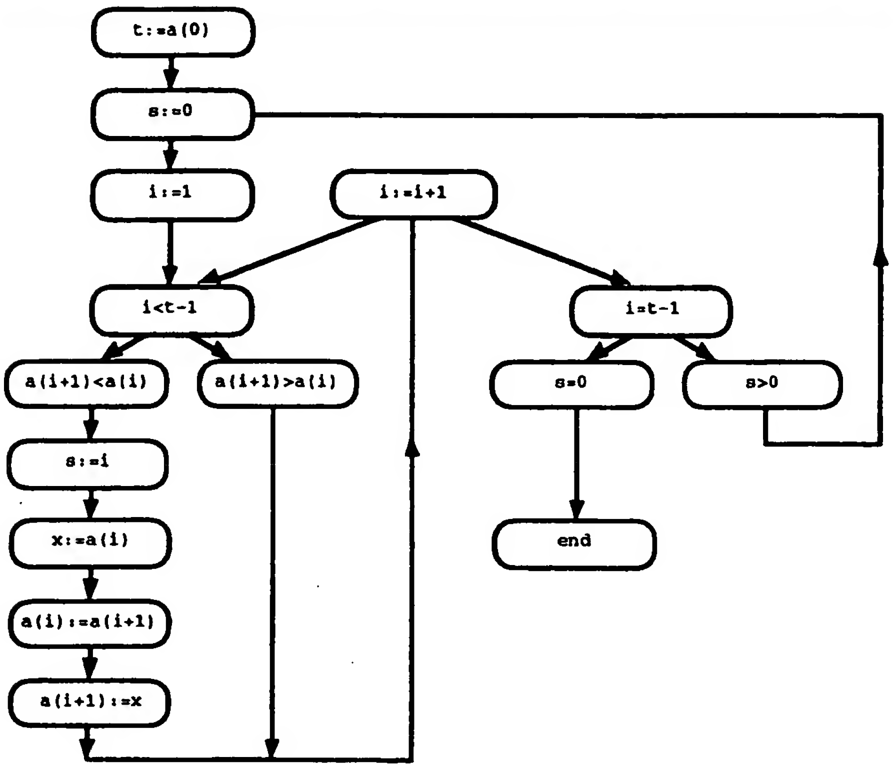


*Figure 2. Program formed from the trace*

## Programming a Calculator by Demonstration

When using interactive computers, there are many situations in which it is hard to decide whether to do a minor, but repetitive, task by hand or write a program to accomplish it. Simple, repetitive arithmetic calculations often present this quandary. For example, to plot $y = xe^{1-x}$ for a dozen or so values of $x$, should one use a hand calculator or write a BASIC program? The first is easier and more certain; it will not take more than 10 minutes. The second may be quicker, but could require consulting the manual to refresh one's memory of the vagaries of BASIC syntax.

The Predictive Calculator is a system that quietly "looks over the shoulder" of the user, forming a model of the action sequence. After a while it may be able to predict what keys will be pressed next. Any that cannot be predicted correspond to "input." Thus the system can eventually behave exactly as though it had been explicitly programmed for the task, waiting for the user to enter a number and simulating the appropriate sequence of key-presses to calculate the answer. The user must pay a price for the service, for the system cannot help but be wrong occasionally; moreover, extra keys must be provided to enable predictions to be accepted or rejected.

**Sample dialogues**

The device is based on the simple, non-programmable pocket calculator shown at the beginning of this chapter. An adaptive model is constructed of the sequence of keys that are pressed. If the task is repetitive (like computing a simple function for various argument values), the system will soon catch on to the sequence and begin to activate the keys itself. When the prediction is wrong an *undo* key was envisaged to enable the user to correct it by undoing a single step of the sequence (though, as noted above, the interface was never completely finished). In order to keep control in the hands of the user, predictions are indicated but not actually made until they have been confirmed by an *accept* key. One can imagine being able to switch into a "free-run" mode after having gained enough confidence in the predictions, but this was not implemented.

Figures 3–5 give examples of this "self-programming" calculator. The first shows the evaluation of $xe^{1-x}$ for a range of values of $x$. The keys pressed by the operator are in normal type; those predicted by the system are shaded. From halfway through the second iteration onwards, the device behaves as though it had been explicitly programmed for the job (except for the need to repeatedly press *accept* or switch into free-run mode). It waits for the user to enter a number, and displays the answer. It takes slightly longer for the constant 1 to be predicted than the preceding operators because the system requires an additional confirmation before venturing to predict a number (see below). Provided the user enters variable data before any fixed constants (and people generally do this), there is no difficulty in getting the calculator to pause when the answer has been reached: it stops automatically whenever it cannot predict the next element in a sequence and this occurs at the end of each of the four calculations in Figure 3.

Figure 4 shows the evaluation of

$$1 + \frac{\log x}{8 \log 2}$$

for various values of $x$. The first line stores the constant log 2 in memory.

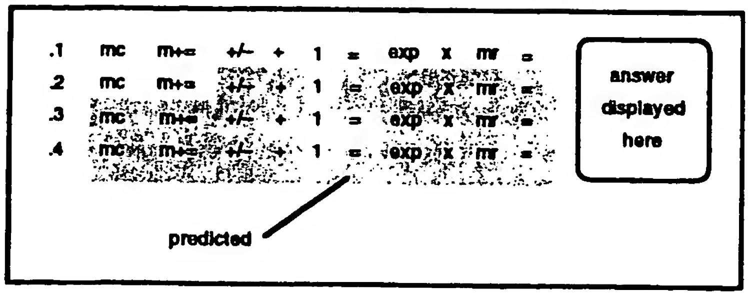

.1 mc m+= +/- + 1 = exp x mr =
.2 mc m+= +/- + 1 = exp x mr =
.3 mc m+= +/- + 1 = exp x mr =
.4 mc m+= +/- + 1 = exp x mr =
answer displayed here
predicted

*Figure 3. Using the self-programming calculator to evaluate* $xe^{1-x}$ *for various values of x*

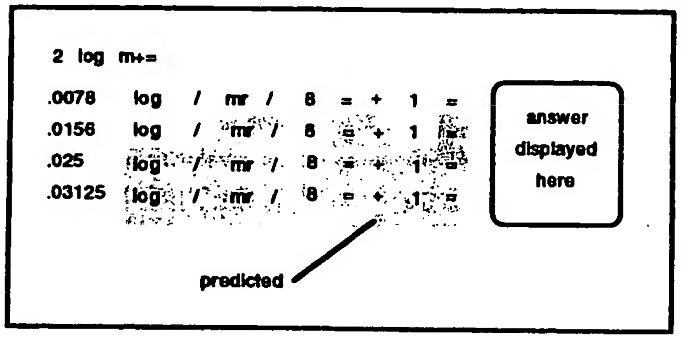

2 log m+=
.0078 log / mr / 8 = + 1 =
.0156 log / mr / 8 = + 1 =
.025 log / mr / 8 = + 1 =
.03125 log / mr / 8 = + 1 =
answer displayed here
predicted

*Figure 4. Evaluating*
$1 + \log x \; / \; 8\log 2$ *for various values of x*

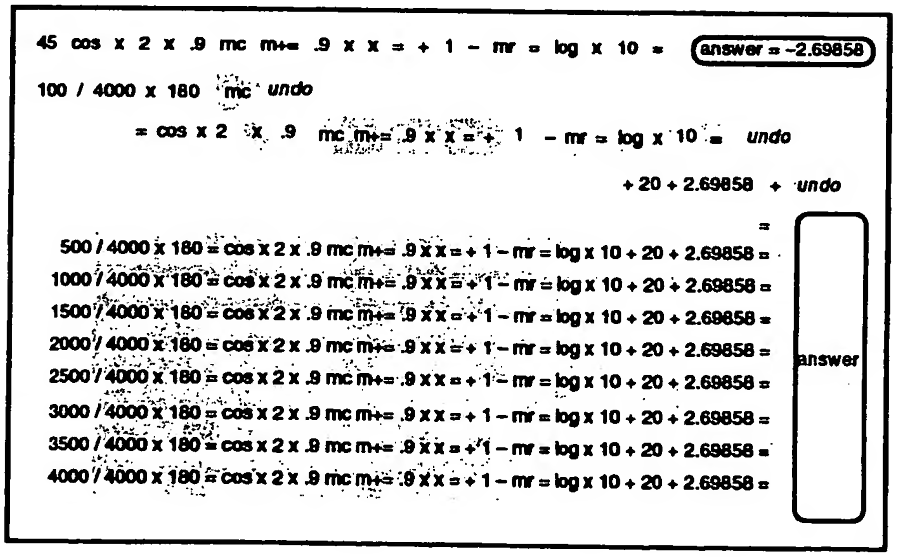

45 cos x 2 x .9 mc m+= .9 x x = + 1 - mr = log x 10 =
answer = -2.69858
100 / 4000 x 180 mc undo
= cos x 2 x .9 mc m+= .9 x x = + 1 - mr = log x 10 = undo
+ 20 + 2.69858 + undo
=
500 / 4000 x 180 = cos x 2 x .9 mc m+= .9 x x = + 1 - mr = log x 10 + 20 + 2.69858 =
1000 / 4000 x 180 = cos x 2 x .9 mc m+= .9 x x = + 1 - mr = log x 10 + 20 + 2.69858 =
1500 / 4000 x 180 = cos x 2 x .9 mc m+= .9 x x = + 1 - mr = log x 10 + 20 + 2.69858 =
2000 / 4000 x 180 = cos x 2 x .9 mc m+= .9 x x = + 1 - mr = log x 10 + 20 + 2.69858 =
2500 / 4000 x 180 = cos x 2 x .9 mc m+= .9 x x = + 1 - mr = log x 10 + 20 + 2.69858 =
3000 / 4000 x 180 = cos x 2 x .9 mc m+= .9 x x = + 1 - mr = log x 10 + 20 + 2.69858 =
3500 / 4000 x 180 = cos x 2 x .9 mc m+= .9 x x = + 1 - mr = log x 10 + 20 + 2.69858 =
4000 / 4000 x 180 = cos x 2 x .9 mc m+= .9 x x = + 1 - mr = log x 10 + 20 + 2.69858 =
answer

*Figure 5. Evaluating a more complex expression*

More complicated is the evaluation of

$$20 + 10 \log \left[1 + a^2 - 2a \cos \frac{180x}{4000}\right] - 10 \log [1 + a^2 - 2a \cos 45]$$

for $a$=0.9, shown in Figure 5. Since the calculator possesses only one memory location, it is expedient to compute the last sub-expression first and jot down the result. The result of this calculation (–2.69858) only has to be keyed twice before the system picks it up as predictable. Some interference occurs between this initial task and the main repeated calculation, and three suggestions had to be "undone" by the user.

If an erroneous suggestion is made, indicated by an "undo," the system is cautious about making a prediction in that context in the future. Any step that is undone is erased from the system's model of the interaction, leaving no record of its existence. Of course, the user will (presumably) have followed the *undo* by an action that is different from the one predicted, creating a conflict in the model. Consequently when the same context occurs again, no prediction will be offered to the user. However, the system will internally monitor its own predictions in that context, and when several correct predictions have been made subsequently (but not presented to the user), it will eventually venture its suggestions. This is the reason why the penultimate "+" on each line of Figure 5 has to be keyed by the user several times. Both of the sequences

```
log  ×  10  =
log  ×  10  +
```

had occurred in the interaction, but a long run of the second was enough to cause it to be used eventually.

As Figure 5 implies, any number that appears in the input sequence is identified and treated as a single unit. To enable the predictions that follow numbers to be made in time for them to be useful, it was necessary to provide a special key with which each number is terminated. Of course, operations like "cos", although written as three letters in the figure, already correspond to a single keystroke and so no terminator is needed.

### The modeling technique

The system uses an extremely simple method to form its predictions. The user's input sequence is characterized by the set of overlapping k-tuples of symbols that occur in it, for some limited context length *k* (*k*=4 in the examples of Figures 3–5). The symbols are either numbers or operation keys. For instance, Figure 6 shows the 4-tuples that are stored for the sequence of Figure 3.

|  |  |  |  |
|---|---|---|---|
| .1 | mc | m+= | +/- |
| mc | m+= | +/- | + |
| m+= | +/- | + | 1 |
| +/- | + | 1 | = |
| + | 1 | = | exp |
| • • • |  |  |  |
| × | mr | = | .2 |
| mr | = | .2 | mc |
| = | .2 | mc | m+= |
| .2 | mc | m+= | +/- |
| • • • |  |  |  |

*Figure 6. The basic model that is stored for the sequence of Figure 3*

The first row shows the first four symbols in the sequence, the second shows the 4-tuple starting at the second symbol, the third starts at the third symbol, and so on. To use these for prediction, whenever the last *k*–1 symbols entered match those that begin a stored tuple, the last symbol of that tuple is predicted. For instance, if the user enters "m+= +/- +" this matches the third row, so the system will predict 1.

This modeling technique, known as a "length-k" modeling [Witten 79], arose out of context modeling techniques [Andreae 77]. It turns out that the k-tuples can be stored economically by massaging them into the form of an automaton model, although this is hardly necessary with modern store sizes. A more powerful, though slightly less space-efficient, prediction technique is to use partial matching on the k-tuples in conjunction with a trie-structured model (e.g. the REACTIVE KEYBOARD [Darragh 92] and the PPM text compression method [Bell 90]); that would be the natural way to implement the predictive calculator nowadays. What is surprising is that the simplistic modeling technique that was actually implemented performed so well in practice.

A modification to the modeling technique was made to suit the calculator situation. It is clear from a cursory analysis of calculator sequences that numbers and operators should be treated differently, for a typical sequence comprises different numbers embedded in a fixed template of operators. This rule is not universal, because fixed constants appear in the stream as well as variable input data. Note how the constants 1 in Figure 3, 8 and 1 in Figure 4, and 4000, 180, 2, .9, 1, 10, 20 and 2.69858 in Figure 5 are all quickly picked out as predictable by the system.

In order to prevent differences in data values from rendering the length-k sequences inoperative, two parallel sets of k-tuples were used. One was exactly as shown above, whereas the other mapped all input numbers into the same token <NUM>, yielding Figure 7 from Figure 6.

Predictions from this model were only used when the other one failed to yield a prediction. For example, when the first prediction of Figure 3 is made, the "+/-" on the second line, it is predicted from the tuple "<NUM> mc m+= +/-", because

| | | | |
|---|---|---|---|
| <NUM> | mc | m+= | +/- |
| mc | m+= | +/- | + |
| m+= | +/- | + | <NUM> |
| +/- | + | <NUM> | = |
| + | <NUM> | = | exp |
| • • • | | | |
| x | mr | = | <NUM> |
| mr | = | <NUM> | mc |
| = | <NUM> | mc | m+= |
| <NUM> | mc | m+= | +/- |
| • • • | | | |

*Figure 7. The mapped model that is constructed from Figure 6*

the actual context "`.2 mc m+=`" has not been seen before. Furthermore, the system was constructed to be more conservative about predicting numbers than operators. No prediction was made unless it would have been correct the previous *n* times it occurred, and *n* was set differently for operators (*n*=1) and numbers (*n*=2). Thus the tuple "`m+= +/- + 1`" is not used to predict the 1 on the second line of Figure 3, but it is used on the third line.

Except for its ability to parse numbers in the symbol string and to distinguish numbers from operators, the system incorporates no knowledge about the calculator or calculation sequences. For example, it will learn nonsense sequences like "`1 1 + + 1 1 + +`" and regurgitate them just as readily as syntactically meaningful sequences. (The sequence "`x x`" that occurs in Figure 5 may seem anomalous; in fact it is the calculator's way of squaring a number.) The procedure is entirely lexical and does not recognize patterns of numbers. For example, in the sequence .1, .2, .3, ... of Figure 3 it is evident what should come next but the system does not spot the pattern. Also, it does not notice multiple occurrences of the same input data. For example, if *x* exp(1–*x*) were to be evaluated on a calculator without a memory, *x* would have to be entered twice. The modeling procedure is incapable of exploiting this redundancy.

A common source of variation in calculator sequences is the discovery of an easier way to do the task. For example, halfway through Figure 5 one may decide to enter 1.81 directly instead of "`.9 x x = + 1`". This is unlikely in the present example, for no penalty is associated with the latter sequence once it has been learned. However, if the simplifying discovery is made early on in the interaction it may cause some incorrect predictions.

## Conclusion

This system illustrates how systematic processing of a "history list" of previous interactions can be used to predict future entries. Rather than being invoked explicitly, the history list is accessed implicitly to provide assistance to the user.

Despite the fact that the modeler has no knowledge of the syntax or semantics of the dialogue, it is remarkably successful. For example, when the three short repetitive problems of Figure 3–5 were concatenated into a single input sequence, over 75% of the sequence elements were predicted and less than 1.5% of predictions were incorrect. There is always a temptation to adjust system parameters in retrospect to yield good performance, and when this was done, nearly 80% of correct predictions were achieved with an error rate of under 0.5%.

The basic idea behind the predictive calculator has been engineered into a device called the REACTIVE KEYBOARD that accelerates typewritten communication with a computer system by predicting what the user is going to type next [Darragh 92]. Obviously, as with the calculator, predictions are not always correct, but they are correct often enough to form the basis of a useful communication device. Since predictions are created adaptively, based on what the user has already typed in this session or in previous ones, the system conforms to whatever kind of text is being entered. It has proven extremely useful for people with certain kinds of communication disability.

## Acknowledgments

It is a great pleasure to acknowledge the influence and encouragement of Brian Gaines, John Cleary, and especially John Andreae. John Darragh skillfully implemented the predictive calculator program on a PDP-11/40 computer.

## Predictive Calculator

Uses and Users

**Application domain:** Calculator

User Interaction

**How does the user create, execute and modify programs?**

User creates programs simply by using the calculator. There is no special learning mode. The only user interaction, beyond doing the task, is to press Undo when the predictor makes a mistake.

By rejecting a prediction, the user forces the calculator to learn a new prediction from that context.

Inference

**Inferencing:**

Keeps a history of all four-token sequences (a token is a number or an operator). Whenever the user types three tokens that match a recorded sequence, the Predictive Calculator predicts the fourth. (Any fixed value can be used instead of "four").

**Program constructs:**

Can ask for user input for any data that varies, but cannot create a variable (e.g. it does not identify the two X's in "$X^2 + 5X$")

Knowledge

**Types and sources of information:**

No syntactic rules. (e.g. it will happily learn the sequence "1 1 + + 1 1 + + " if that is what the user enters)

Implementation

**Machine, language, size, date:** PDP-11/40, 1982.